AFIT/GCS/ENG/91D-21

AD-A243 622

DTIC
ELECTE
DEC 20 1991
C

Object-Oriented Analysis and Design of the Saber Wargame

THESIS

Christine M. Sherry
Captain, USAF

AFIT/GCS/ENG/91D-21

91-18988

Approved for public release; distribution unlimited

91 12 24 029

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
Object-Oriented Analysis and Design of the Saber Wargame

5. FUNDING NUMBERS

6. AUTHOR(S)
Christine M. Sherry, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-21

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Wargaming Center, Maxwell AFB, AL 36112-5532

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Abstract

This thesis presents an object-oriented analysis and design of Saber, a theater-level computerized wargame, for the Air Force Wargaming Center, Maxwell AFB, Alabama. The analysis and design is based on a recently developed conceptual model, an existing land battle, and additional research. This thesis also begins the implementation process.

The design was accomplished using an iterative, five step design process. Objects and operations were chosen and then encapsulated in Ada packages. This thesis also makes necessary changes to the land battle as described by the conceptual model and as the result of additional research.

Sound software engineering principles were used to ensure that the system is easily modified or enhanced. Once Saber is completely implemented it will provide a wargame that is both flexible and credible, due in part, to the fact that it is not tied to a specific theater or to specific forces.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Oriented Design, Wargame	127
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT/GCS/ENG/91D-21

Object-Oriented Analysis and Design of the Saber Wargame

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

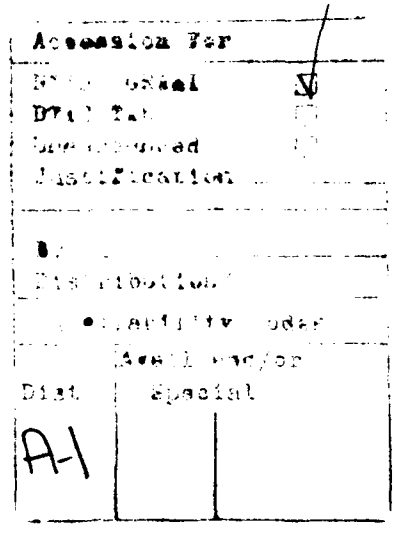
In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Christine M. Sherry, A.A.B., B.A., M.S.S.M.

Captain, USAF



December, 1991

Approved for public release; distribution unlimited

Acknowledgments

I would like to thank my thesis advisor, Maj Mark Roth, and my committee members, Maj Michael Garrambone and Lt Col Patricia Lawlis, for their assistance and support in the development of this thesis. I would also like to thank the following fellow students: Capt Gary Klabunde, who shared his technical expertise, and Capt Dennis Rumbley, who provided moral support during the past year.

I also owe my thanks to my friend, Nina Stutler, who patiently listened to me talk about the thesis for the past year. As always, I also owe my thanks to my parents, Mr. and Mrs. Gervase J. Sherry, who are always there when I need them. Lastly, and most importantly, I owe thanks to God, for with Him all things (including this degree) are possible.

Christine M. Sherry

Table of Contents

List of Figures

Abstract

This thesis presents an object-oriented analysis and design of Saber, a theater-level computerized wargame, for the Air Force Wargaming Center, Maxwell AFB, Alabama. The analysis and design is based on a recently developed conceptual model, an existing land battle, and additional research. This thesis also begins the implementation process.

The design was accomplished using an iterative, five step design process. Objects and operations were chosen and then encapsulated in Ada packages. This thesis also makes necessary changes to the land battle as described by the conceptual model and as the result of additional research.

Sound software engineering principles were used to ensure that the system is easily modified or enhanced. Once Saber is completely implemented it will provide a wargame that is both flexible and credible, due in part, to the fact that it is not tied to a specific theater or to specific forces.

Object-Oriented Analysis and Design of the Saber Wargame

I. Introduction

This thesis effort is based on two previous Air Force Institute of Technology thesis efforts. CPT Marlin Ness, a graduate student in engineering, developed and implemented a land battle module for a computer wargame. CPT William Mann, an operations research graduate student, developed a conceptual model which extended CPT Ness' land battle and added an air module. The resulting new model is called Saber. The model creates a "foundation for a new theater level computerized wargame for the Air Force Wargaming Center" located at Maxwell AFB, AL (17:1).

This thesis effort performed an object-oriented development of the previous theses. It first took Mann's conceptual model and conducted an object-oriented design. The design was conducted to ensure compatibility with CPT Ness' earlier work. It then began the implementation using the Ada programming language. Minimal modifications were made to Ness' code.

1.1 Background

CPT Ness' land model is an "aggregated interactive theater-level land combat model" which was implemented using the Ada programming language and object oriented design techniques (20:x). The land battle model "simulates doctrinal planning and decision making operations conducted at Army Group level" (20:x). The model has the capability of playing various scenarios (17:5) and contains modules to simulate logistics, unit movement and attrition, as well as intelligence (20:x). Appendix A further describes CPT Ness' land module.

CPT Mann's thesis problem was to link, "US Air Force doctrine with a conceptual model's framework and designs a new air battle module" (17:5). The new combined land and air modules will be called Saber (17:5).

Saber consists of many modules, including air-to-air, air-to-ground, ground-to-air, logistics, intelligence, nuclear, and chemical weapons (17:8-9). The model takes input from the wargame players, simulates the war, and provides output in the form of status and combat results. The output from one run is used as input for the next run (17:53).

The simulation of the war follows "a decrement process that expends resources, aircraft, and ground forces as the model represents battle" (17:53). Appendix B further describes CPT Mann's conceptual model.

1.2 Problem Definition and Research Objective

The Air Force Wargaming Center needs a viable and flexible computer wargame for students at the Air War College to use as a learning tool. Ness' thesis effort began this process, Mann's conceptual model continued the effort. The purpose of this thesis effort is to combine Ness' prototype land battle model with Mann's conceptual air battle description to build an integrated theater level computer wargame.

1.3 Scope and Limitations

This thesis effort creates a prototype version of an integrated theater level land and air computer wargame. The scope of this thesis does not include designing or implementing the databases which are needed to maintain information on the various weapons, aircraft, etc. This thesis does not include defining the contents of the databases. Capt Andre Horton's thesis effort was to perform an object-oriented database design for Saber. Capt Horton's database produces flat files for input to Saber. (11)

The thesis does not include designing or implementing any graphical interfaces to the wargame. The design and implementation was coordinated with Capt Gary Klabunde. Capt Gary Klabunde's thesis effort was to design a graphical interface for Saber. (13)

The thesis does not include designing the input screens or the output screens and reports. Horton's thesis work included designing the input screens; Klabunde's included designing the output screens and the combat reports. (11, 13)

Figure 1 depicts the portions of a typical combat model as portrayed in Mann's thesis. (17:7) Horton's thesis work concentrated on the preprocessor portion, Klabunde's on the postprocessor, and this thesis on the simulation itself.

This thesis effort does not include conducting any sensitivity analysis or validation and verification nor does it include the writing of any scenarios.

1.4 Assumptions and Constraints

The following assumptions and constraints apply:

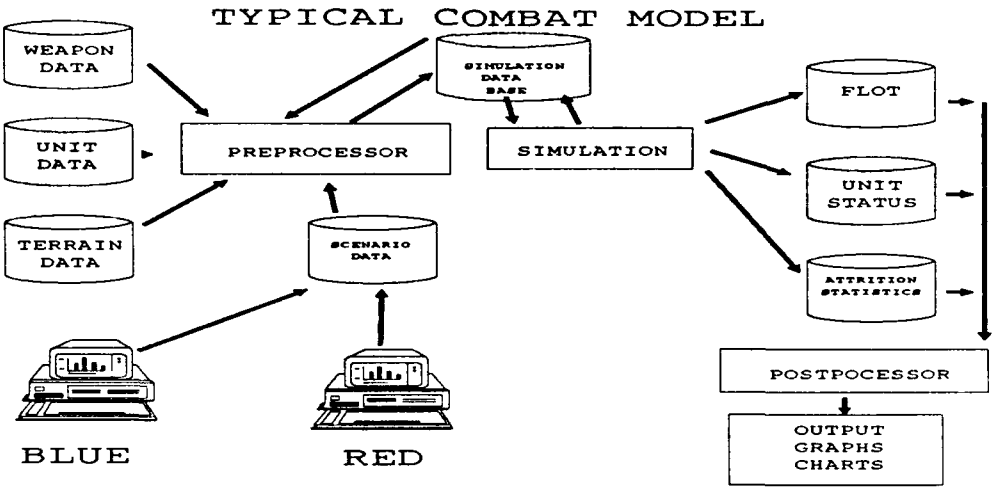


Figure 1. Typical Combat Simulation Model

- The land model, as implemented by Ness, adequately models ground war to the level desired for the training requirement.
- The mathematical formulas outlined in Mann's thesis for the air model are adequate for the purpose intended.
- Mann's research on both U.S. and U.S.S.R. air and land doctrine provided him with accurate mission information.
- At this time, Saber simulates only land and air combat operations. It does not simulate naval operations.
- Saber is an unclassified, aggregated theater-level wargame.
- "The new game will have the flexibility to play any scenario or in any theater of operations" (17:6).
- "Command, control, and communications will be modeled by the player interaction in the game and not by the computer simulation" (20:5).

- The Air Force Wargaming Center is responsible for verification and validation of Saber.
- Concurrent thesis efforts will provide the graphical interface and database support for the wargame.
- The DOD standard programming language, Ada, will be used to implement the wargame.

1.5 Methodology

The beginning point for this effort was Mann's thesis which was analyzed using object-oriented requirements analysis and design techniques. As part of the design phase, the fields of the various databases were defined and agreed upon with the student working on the concurrent thesis effort for database support. The object-oriented design was followed by an Ada implementation. This implementation follows sound software engineering principles. The actual program code is documented to permit easy maintenance by the Air Force Wargaming Center.

1.6 Materials and Equipment

The Air Force Wargaming Center expects to receive Sun SPARC workstations; therefore, the modification of the land module and the start of the implementation of the air module was accomplished on a Sun SPARC workstation which was already available at AFIT. Version 6 of Verdix Ada was used for code development.

1.7 Expected Benefits of This Research

This thesis is the second in a series of thesis efforts to develop a new computer wargame for the Air Force Wargaming Center. Saber, once completely implemented, will provide the Air Force Wargaming Center with a flexible and viable computer wargame. This wargame will be used by students at the Air War College to aid in their learning of the wartime operations of planning, preparation, and execution.

1.8 Thesis Overview

Chapter II consists of a literature review of object-oriented design methods and simulations with an emphasis on using object-oriented techniques in simulations. Chapter

III describes both the process used for the requirements analysis and the initial design. It also discusses the results. Chapter IV discusses the modifications and enhancements to Saber which resulted from wargaming research group meetings, as well as clarifications on material in Mann. Chapter V describes the detailed object-oriented design. Chapter VI discusses the implementation phase of Saber. Chapter VII contains a summary and conclusion as well as recommendations for future work.

II. Literature Review

2.1 Introduction

The purpose of this chapter is to review some of the literature on object-oriented design and simulation. First, object-oriented design concepts are discussed, including the steps involved in conducting an object-oriented design as described by various authors. Advantages and strong points of object-oriented design are also discussed.

Finally, simulation is described, including continuous and discrete event simulation. Since Ada will be used to implement Saber, research also consisted of tying discrete event simulation to an Ada implementation. This chapter also includes a brief discussion of using object-oriented design to develop simulations.

Some authors refer to "object-oriented design" or development while others do not hyphenate the words "object" and "oriented". For the sake of consistency, this chapter hyphenates the words even when the author quoted did not hyphenate them. As with the term object-oriented, the term "discrete event" is sometimes hyphenated. For the sake of consistency, this thesis uses the unhyphenated version of the words whether the author quoted used the hyphen or not.

2.2 Object-Oriented Design

Object-oriented design (OOD) is based on a decomposition of the system into objects. This differs from functional decomposition techniques where the decomposition is based on functions. Each module in an object-oriented design is based on an object whereas, in the functional decomposition, the modules are based on steps in the overall system process (1:211). "Object-oriented design is a design method which is based on information hiding" (28:204). Korson and McGregor state that "the object-oriented design paradigm takes a modeling point of view" (15:46).

2.2.1 Object-Oriented Design Concepts. Korson and McGregor describe five concepts in object-oriented methods. These concepts, which are described in the following sections, are: "objects, classes, inheritance, polymorphism, and dynamic binding" (15:42).

2.2.1.1 Objects. Booch defines an object as "something you can do things to. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable"

(4:516). The behavior of an object is "characterized by the actions that it suffers and that it requires of other objects" (1:211). "The intent of an object is to represent a problem domain entity" (25:4-57).

Objects use memory and have an associated address. Associated with an object are procedures and functions which define the operations on the objects. (15:42) "Objects communicate by passing messages to each other and these messages initiate object operations" (28:204). Communication may be asynchronous. OOD is an excellent method to use in designing parallel or sequential programs. (28:204)

2.2.1.2 Classes. A class is "a set of objects that share a common structure and a common behavior. The terms class and type are usually (but not always) interchangeable; a class is a slightly different concept than a type, in that it emphasizes the importance of hierarchies of classes" (4:513). "From the point of view of a strongly typed language, a class is a construct for implementing a user-defined type" (15:42).

Object-oriented techniques use an Abstract Data Type (ADT) to represent a class of objects. According to Booch, an ADT "denotes a class of objects whose behavior is defined by a set of values and a set of operations, including constructors, selectors, and iterators" (2:613). "Ideally, a class is an implementation of an ADT. This means that the implementation details of the class are private to the class" (15:42). An ADT in Ada is implemented by using the package construct. A "package encapsulates the type but is not the type itself" (15:42). This "results in a weaker connection between state and behavior" (15:42).

2.2.1.3 Inheritance. "Inheritence is a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes" (15:43). "Inheritance defines a 'kind of' hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses" (1:514). Korson and McGregor state that the inheritance relation often denotes an "is a" relation. Inheritance supports reuse of software components. (15:43-44)

2.2.1.4 Polymorphism. Polymorphism is defined as "a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways" (4:517).

In other words, this means that polymorphism is a technique in which an object can have more than one form. "A polymorphic reference has both a dynamic and a static type associated with it. The 'is a' nature of inheritance is tightly coupled with the idea of polymorphism in a strongly typed object-oriented language" (15:45).

2.2.1.5 Dynamic Binding. Booch defines dynamic binding as "a binding in which the name/class association is not made until the object designated by the name is created (at execution time)" (4:513). Binding, as defined by Booch, "denotes the association of a name (such as a variable declaration) with a class" (4:513). Korson and McGregor state that dynamic binding "means the code associated with a given procedure call is not known until the moment of the call at runtime" (15:46). Dynamic binding "is associated with inheritance and polymorphism in that a procedure call associated with a polymorphic reference may depend on the dynamic type of that reference" (15:46).

2.2.2 Object-Oriented Design Process. Different authors describe different steps to use in conducting an object-oriented design. What one author calls an object-oriented design, another author calls object-oriented development or requirements analysis. Since many authors use a modified version of Booch's object-oriented design process, his steps will be discussed in a later section of this chapter. This section discusses a method described by Henderson-Sellers and Edwards which they call an object-oriented development methodology.

Henderson-Sellers and Edwards describe seven steps used by Bailin in his object-oriented requirements specification method. They state that these steps could "obviously transcend the requirements stage well into detailed design" (10:148). Bailin's seven steps, as described by Henderson-Sellers and Edwards, are (10:148-149):

1. identification of key problem space objects,
2. distinguish between active and passive objects,
3. establish data flows between active objects,
4. decomposition of objects into "sub-objects",
5. check for new objects,
6. group functions under new objects,
7. assign new objects to appropriate domains.

According to Henderson-Sellers and Edwards, Bailin sees the first three steps as ones which are accomplished only once, while the other steps are performed iteratively. Henderson-Sellers and Edwards propose a "seven-point methodological framework for object-oriented systems development" (10:149). The steps, and a description of each follow (10:140-150):

1. Undertake object-oriented systems requirements specification. "This stage is a high-level analysis of the system in terms of objects and their services, as opposed to the system functions" (10:149).
2. Identify the objects and the services each can provide. This equates to the entities and their interfaces. "This is where the functional features will be defined; although no indication of implementation is required" (10:150). Henderson-Sellers and Edwards propose that an object dictionary be established. The visible interface is defined by identifying the objects, and the operations on the objects, as well as the services offered.
3. Establish interactions between objects in terms of services required and services rendered. Henderson-Sellers and Edwards suggest that an entity-data flow diagram (EDFD) or entity-relationship diagram be used for this step. They suggest that a better name for this diagram is an information flow diagram (IFD).
4. Use of lower-level IFDs. This is where analysis and design merge. The lower-level IFDs show "more internal details of the objects" (10:150). From this step on, bottom-up concerns should be analyzed.
5. Bottom-up concerns. During this step, objects are constructed from libraries of previously used objects. Implementation of low-level classes begins.
6. Introduce hierarchical inheritance relationships as required. This step involves determining whether there are any superclasses or new subclasses. Henderson-Sellers and Edwards propose the use of an inheritance diagram to show the inheritance relationships. They state that this step is needed to provide a well-defined hierarchy so that future efforts can reuse the resulting structure.
7. Aggregration and/or generalization of classes. This step might require reviewing and modifying the IFDs. Prototyping might begin at this stage. The identified system classes can undergo another stage of development which Henderson-Sellers and Edwards call generalization. "At this stage the components continue to be worked on until they are general, generic, and robust enough to be placed in a library of components" (10:150).

2.2.3 Booch's Object-Oriented Design Process. This section describes the five steps of Booch's design process as described in his book, *Software Components with Ada* (2). Since other authors use very similar steps, it includes information from various authors.

2.2.3.1 Identify the Objects and Their Attributes. This step involves taking a narrative requirements document and extracting the nouns, pronouns, and noun phrases (2, 8, 12). Some objects may be similar to other objects.

In this case, a class of objects is formed (3:48). Once all the objects and classes are identified, a decision must be made as to whether they will be kept or discarded (12:44). Just because an object is identified from the requirements document does not mean that it should become part of the design and implementation (12:44).

Once the list of objects is refined, then the attributes of the objects should be determined. "The attributes of an object characterize its time and space behavior" (2:17). Jean and Strohmeier state that "these properties are given by the qualifiers of the objects and classes within the informal strategy and by the additional information found in the requirements analysis document" (12:44). EVB Software Engineering, Inc. states that these are the "adjectives and adjectival phrases" (8:2-8).

2.2.3.2 Identify the Operations Suffered By and Required of Each Object. In this step, the requirements document is used to extract verbs, verb phrases, and predicates (8, 12). Then, the extracted verbs, verb phrases, and predicates are associated with a particular object (8, 12). Jean and Strohmeier say "The goal is to bind each operation to a single object or a single class" and that "no operation should be left alone" (12:45).

"The operations suffered by an object define the object's activity when acted upon by other objects". By defining the operations required by an object, an attempt is made to decouple objects from one another. (2:17)

During this step, a determination should be made as to whether the operation is a selector, a constructor, or an iterator (12:45). A selector evaluates the current object state; a constructor alters the state of an object; an iterator permits all parts of an object to be visited (2:20).

2.2.3.3 Establish the Visibility of Each Object in Relation to Other Objects. As part of this step, a decision is made as to what objects "see" and are "seen" by other objects (3:49). The dependencies among objects should be established (1:219). This can

be done diagrammatically by drawing each object and then connecting the objects with a line to show the visibility between the objects (2:28,30).

EVB divides Booch's step into four substeps. The first substep is to decide on how to implement the operations. Subprograms, packages, tasks, and generics are the Ada program units used to implement an object. The second substep formally describes the interfaces among the objects. These descriptions can be textual or graphical. A program unit which depends on another program unit must be compiled after the first program. This substep helps determine the compilation order. EVB's third substep is to create any additional objects and operations which are needed to help the implementation strategy. These items are ones that were not identified as part of the informal strategy but must be visible outside of the program unit. The last substep is to produce graphical annotations to represent the formal strategy. The diagrams give no indication as to how an object should be implemented nor do they show much about the underlying implementation of the operations. The diagrams serve as a map for the software engineer to follow throughout the rest of the design process. (8).

2.2.3.4 Establish the Interface of Each Object. This step is accomplished by writing a module specification for each object. This can be done in Ada by producing an Ada specification which can be compiled. Booch states that "this specification also serves as a contract between the clients of an object and the object itself". (2:18)

2.2.3.5 Implement Each Object. This "involves choosing a suitable representation for each object or class of objects and implementing the interface from the previous step" (2:18). An object is implemented in Ada "as a packaged set of procedures and internal data" (25:4-55).

2.2.4 Advantages and Strong Points of Object-Oriented Design. Sommerville describes the following advantages to OOD (28:205):

- Message passing eliminates the need for shared data areas for communication between objects. Overall system coupling is thus reduced.
- All state and representation information is kept within the object itself, making the object an independent entity that may be readily changed. Objects can not access information on other objects either deliberately or accidentally. Changes may be made without reference to other system objects.

- Objects may execute either in parallel or sequentially. They may also be distributed. The decision as to whether parallelism should be used does not need to be made at an early stage of the design process.

Korson and McGregor describe seven ways in which object-oriented design provides support for a good design.

1. Modularity. Classes become the modules. "This means that not only does the design process support modularity, but the implementation process supports it as well through the class definition". (15:50)
2. Information Hiding. "The class construct supports information hiding through the separation of the class interface and the class implementation" (15:51). This separation permits the class specification to be mapped to various implementations and means some maintenance can be accomplished without the user's knowledge (15:51).
3. Weak Coupling. Object-oriented design supports weak coupling (15:51). Since classes are designed as a collection of objects and the operations on those objects, the "interface operators of a class are inward-looking in the sense that they are intended to access or modify the internal data of the class" (15:51). This leads to less coupling which is desirable.
4. Strong Cohesion. Strong cohesion is desirable and supported by object-oriented design. Korson and McGregor state that "a class is a naturally cohesive module because it is a model of some entity" (15:51). Functional cohesion is a desirable form of cohesion. Booch defines it as cohesion "in which the elements of a class or module all work together to provide some well-bounded behavior" (4:124). OOD supports functional cohesion. The fact that OOD makes use of inheritance does not mean that the cohesion is weakened since both the data and functions which are inherited from another class form a natural group (15:51). These natural groups are "brought together to represent one concept" (15:51).
5. Abstraction. Object-oriented design supports abstraction. Booch defines abstraction as "the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply-defined conceptual boundaries relative to the perspective of the viewer" (4:512). Korson and McGregor discuss two types of abstraction which support OOD: abstraction by specification and abstraction by parameterization (15:51-52). Abstraction by specification separates the specification

of an object from its implementation (15:52). Abstraction by parameterization abstracts the type of data to be manipulated from the specification of how it is to be manipulated" (15:52). Seidewitz and Stark state that there is a "spectrum of abstraction" including entity, action, virtual machine, and coincidental abstraction, which in conjunction with information hiding, provide the main guidance for defining objects (25:4-57). Entity abstraction, which is the best level, is where an object "represents a useful model of a problem domain entity" (25:4-57). Action abstraction is where "an object provides a generalized set of operations which all perform the same kind of function" (25:4-57). Seidewitz and Stark describe virtual machine abstraction as the case in which "an object groups together operations which are all used by some superior level of control or all use some junior level set of operations" (25:4- 57). The worst level of abstraction is the coincidental. This level of abstraction is defined as where "an object packages a set of operations which have no relation to each other" (25:4-57).

6. Extensibility. Object-oriented methods are "easily extended" (15:52). Inheritance supports this in two ways. First, because inheritance permits "the reuse of existing definitions to ease the development of new definitions" (15:52). Second, the polymorphic property also supports extensibility in designs (15:52).
7. Integrable. Designs produced by OOD "facilitate the integration of individual pieces into complete designs" (15:52). This includes both the use of classes and objects (15:52).

Booch discusses coupling, cohesion, sufficiency, completeness, and primitiveness as means of determining that a design is good. Coupling and cohesion were discussed above. By sufficiency, Booch "means that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction" (4:124). Completeness means "that the interface of the class or module captures all of the meaningful characteristics of the abstraction" (4:124-125). Completeness is a subjective matter and should not be overdone. Primitiveness implies that an operation can only be implemented if the developer is given access to the underlying representation of the ADT. (4:124-125)

2.3 Simulation

"Simulation involves constructing models and the study of model behavior within varying environments" (30:181). One of the reasons to use simulation is to experiment

with as yet undefined systems (30:181). Another reason to use simulation is because first hand experience is not available (30:181). "Simulation is one of the more widely used techniques in managerial decision making" (7:83). A simulation can focus on the important details of what it is modelling and leave out the less important details (30:181). Unger and others state that "the problems associated with designing, implementing, and maintaining large simulation programs are essentially identical to other types of programs, particularly concurrent programs" (30:181). Objects, and their operations, can readily be represented in simulations (30:182).

The following subsections describe continuous and discrete event simulations. Since the wargame which will result from this thesis effort will use discrete event simulations and Ada the discussion includes a description of discrete event simulation and the Ada programming language.

2.3.1 Continuous Simulation. "Continuous simulations are used in the modeling of systems in which changes in system state variables occur continuously over time" (30:238). This type of system is described by using "time-dependent differential or difference equations that represent the rates of change of the system's state variables" (30:238). With continuous simulations, "time is advanced in uniform, or constant size, steps" (30:181).

According to Unger and others, continuous simulation methods are used in a number of applications including "the modelling of marine and aerospace navigation and control systems" (30:238).

2.3.2 Discrete Event Simulation. "Discrete event simulation refers to a modelling technique that enables changes to occur in the state of a model at arbitrary simulation times" (30:181). The actual time when the changes occur is called an event. The time intervals between events are usually not of the same duration. (30:181) Discrete event simulations are used as analytical tools in the investigation of complex systems (5:5-105).

2.3.2.1 Discrete Event Simulation and Ada. According to Borrego and others, "Ada is a general purpose programming language and does not provide any simulation tools" (5:762). In constrast, according to Shtern, Ada has many characteristics, like strong typing and private types, which make it a powerful simulation language (27:13). Two other Ada features, which discrete event simulations should take advantage of, are generic packages and tasks (5, 27). Generic packages serve as templates for a package (5:762). Tasks are used to represent the events and processes in the system being implemented

(26:5-105). The tasks can run independently on a single processor computer or in parallel on a multi-processor computer (21:8). Communication between the tasks is achieved by using shared variables and rendezvous parameters (21:8). The rendezvous in Ada provides the mechanism which ensures that while one task is being run, it is "nonsharable and nonpreemptable for use by another task" (21:8). Tasks can be used in implementing simulations because they can be suspended and resumed (5:762).

Melde and Gage discuss the development of a generic Ada package, A*SIM, which provides all the capabilities needed for "general-purpose discrete event simulation". Besides discussing how they used Ada's generic packages and tasks to implement the simulation, they also discuss other Ada features which they found helpful. For example, they used the Ada record to provide "dynamic allocation and deallocation of blocks of memory for data storage". They also found that access types permitted them to create complex data structures like linked lists. In A*SIM, the event calendar implementation is designed so that it "can vary from a simple linked list structure to a highly specialized data structure and algorithm for efficiency". (18:58-60)

Borrego and others describe some problems they found when using Ada to implement a simulation. One problem with using Ada tasks is that the tasks are implemented by the use of a first-in-first-out queue, unless a priority is provided. Priorites are determined at compile time. There is no method for the simulation, based on various data, to change the priority during run time. "Discrete event simulation also requires a random number generator". A random number generator is not a required feature of Ada. (5:762)

In contrast, Shore describes the following benefits of using Ada in implementing a simulation (26:5-105):

- Using Ada does not require training programmer personnel in more than one programming language. "After some training in the concepts of discrete event simulations, an Ada programmer could create simulations in a familiar environment, rather than requiring additional training in a completely new language".
- Using Ada also permits easy porting of the simulation to any computer which has an Ada compiler.

2.4 Object-Oriented Design and Simulation

"The object-oriented design of simulations is based on the concept of abstract data types" (29:123). Object-oriented techniques lend themselves to simulation because the

"things" which should be modelled are objects and what each of the "things" can do are the operations on the objects (23:278). This defines an abstract data type. Roberts and Heim state that an "object-orientation attempts to bridge the gap between the model and what is modeled" (23:278). They also state that "division into classes, recognition of methods, and the organizations of hierarchies form the basic approach to object-oriented modeling" (23:279). Methods are the operations performed on an object.

One benefit of an object-oriented simulation system is the focus on objects. Focusing on objects provides both data abstraction and information hiding which help to modularize the system. This "stimulates the user to identify the principal components of a system and to specify their behaviors and interactions". (23:279)

Another benefit of an object-oriented simulation is that existing models can form the basis for new models. By using overloading and inheritance, old objects can take on new meanings. (23:280)

The resulting amount of code generated using object-oriented simulations is less than using traditional approaches. This makes it easier to manage the model and also permits models to be larger and more realistic. (23:280)

Objects provide a natural starting point for concurrency (23:280). Concurrency permits more than one object to be processing at the same time as long as the objects do not need to communicate with each other.

2.5 Summary

This chapter discussed object-oriented design and simulation. Object-oriented design was defined and various concepts described. The key terms in object-oriented techniques are object and class. An object is something which can be changed. It has behavior, state, and identity. When objects have a similar structure and behavior they are often grouped into classes. Another term defined was inheritance. Ada, which was used to implement Saber, does not support inheritance.

Two object-oriented design processes were described, including an in-depth description of Booch's process. The research conducted showed that the process, as described by numerous authors, is basically the same. The first step is to identify the objects and group them into classes. At the same time as objects are identified, the operations which those objects require can also be determined. As with other design techniques, an object-oriented design process should be an iterative one.

This chapter also described some of the advantages and strong points of object-oriented design. Object-oriented design techniques provide the implementer with an easy way to follow sound software engineering principles. Object-oriented design techniques provide a modularized system which permits easier maintenance of the actual code. It also supports information hiding by separating the interface and the implementation permitting some maintenance to take place without the user's knowledge. A good object-oriented design ensures weak coupling and strong cohesion as well as supporting abstraction. These are all very important software engineering principles.

Simulation was the second major topic to be discussed in this chapter. Simulation is the constructing of models and the study of model behavior within different environments. There are two types of simulation, continuous and discrete event. Continuous simulations are used to model systems where the variables change continuously over time. Discrete event simulations are used to model systems where the variables change at arbitrary times. Since wargames are discrete event simulations, research concentrated mostly in that area. Object-oriented design techniques lend themselves to discrete event simulations because the variables in a simulation can be modelled as objects and the events which change variables become the operations on the objects.

Some researchers thought that Ada does not provide the capabilities needed to implement a simulation. Other authors expounded on how Ada can be used for simulations. The Ada task construct provides the capability for the simulation to suspend and resume a task if needed. Other features of Ada which make it a good implementation method are: the ability to encapsulate objects into packages, generic packages, strong typing, and private types.

The third major topic in this chapter briefly described using object-oriented design techniques in developing a simulation. It also discussed some of the benefits gained in using object-oriented design.

The next chapter describes the requirements analysis and design phase which was accomplished as part of this thesis effort.

III. Requirements Analysis and Initial Design

3.1 Introduction

This chapter describes the results of the requirements analysis and initial object-oriented design. The requirements analysis was accomplished by analyzing both Ness' and Mann's theses. The object-oriented design was only accomplished on Mann's thesis. The design phase consisted of determining the objects, their attributes, and the operations needed.

3.2 Requirements Analysis

As part of the requirements analysis, a review of CPT Ness' code, thesis, and maintenance manual was accomplished. The goal at this stage of the development was not to become familiar with how functions were accomplished but rather how they were organized. The Ada packages and procedures were shown diagramatically in order to clearly and quickly see the relationships between the individual modules.

The second part of the requirements analysis was to become familiar with CPT Mann's thesis. An attempt was made to extract the basic information needed in order to begin the development process. Further information on Ness' and Mann's theses can be found in Appendices A and B, respectively. Appendix C shows the packages Ness used. It lists the procedures and functions which are part of each package as well as giving a brief description of his main type declarations. Appendix D shows the visibility required between Ness' original packages.

3.3 Initial Object-Oriented Design

The first three of Booch's object-oriented design steps, as discussed in Chapter II, were used to conduct the initial object-oriented design.

3.3.1 Identify the Objects and Their Attributes. The first step in this process is to determine the objects. This is accomplished by extracting nouns from the requirements document. In this case, both Ness' and Mann's theses were used as requirements documents. The initial list of objects from the requirements analysis was refined as part of this step and the attributes determined. The emphasis during this step was on the air battle, though the land battle was considered. The objects and corresponding attributes are shown in Appendix D. Items shown in italics were not part of the initial design.

An example of an object is an aircraft. An aircraft object would need many attributes in order for the simulation to function correctly. It would need, for example: the plane's designation, the weapons load which it carries, the average number of sorties flown per week, the maximum speed of the plane, the amount of maintenance it requires each time it is flown, and the size of runway needed. Other objects that come to mind immediately are Weapons and Bases.

3.3.2 Identify the Operations Suffered By and Required of Each Object. The operations are determined by extracting verbs and verb phrases from the requirements documents. For the initial design, this step was only accomplished using Mann's thesis. The objects and the required operations are shown in Appendix E. Items shown in italics were not part of the initial design.

To continue with the Aircraft object, it would need a number of operations. The simulation would need to read in from a disk file the initial characteristics for each type of aircraft. It would also need to determine the actual maintenance hours which a plane requires based on the mission flown. The simulation would also need to keep track of the number of aircraft available, decreasing the quantity when a plane is on a mission and increasing it when the plane returns from a mission.

3.3.3 Establish the Visibility of Each Object in Relation to Other Objects. The purpose of this step is to decide which objects need to "see" and be "seen" by other objects. Appendix F shows the initial results of this step. The objects marked with an "I" indicate the results of this step.

Since an aircraft needs to have a weapons load attribute, the Aircraft object would need access to the Weapons object. And since an aircraft is assigned to a base, it should be "seen" by the Base object. The Weapons object would also be "seen" by the Base since a base is a holder of weapons as well as aircraft.

3.4 Summary

This chapter briefly described the steps taken to accomplish both a requirements analysis and the initial object-oriented design. Object-oriented design is a good technique to use for simulations, especially wargames, because the objects are real "things", like weapons and aircraft. Unfortunately, there is also one major drawback: the need to extract nouns (for objects) and verbs (for operations) from the requirements document.

This can be very time consuming if the requirements document happens to be over 200 pages long. The process could be automated if a dictionary of valid nouns and verbs was available or constructed and if an electronic version of the requirements document was also available. Once the list of nouns and verbs are made, an analysis must be accomplished to determine if the words are actually objects and operations needed for the system being designed.

The next chapter will discuss changes to the requirements which resulted from thesis research meetings.

IV. Requirements Modifications, Clarifications, and Enhancements

4.1 Introduction

This chapter will describe changes to features that were implemented by Ness or were part of Mann's thesis effort. It will also include features that were not defined by either Ness or Mann but were defined as part of the research group meetings and discussions with the Air Force Wargaming Center.

The sections are centered around whether the discussion affects Saber as a whole, only the land module, or only the air module.

4.2 Saber

4.2.1 The Hex System. Ness' hex system numbered the hexes from west to east and south to north along a diagonal. There were two coordinates, one representing the x-coordinate and the second representing the y-coordinate. The x-coordinate increases as you move from west to east and the y-coordinate changes with diagonal movement. The hexes were oriented with north-south directions across the points and east-west directions across the flats. Figure 2, which was extracted from Ness' thesis, shows his basic hex setup and numbering scheme. (19:8-9)

As part of the graphical interface, the hexes should be displayed. With Ness' hex grid, it is not easy to determine from a display what the hex number is. The Saber research group reviewed a BDM document which described various types of grids and different numbering schemes (16). The members kept two areas in mind: readability of the display and the need to be able to determine the relationship between seven land hexes and one air hex. It was decided that Ness' numbering scheme should be modified so that the x-coordinate increases moving from west to east and remains the same when moving north or south on the grid. The y-coordinate remains the same when moving from west to east and changes when moving north or south on the grid.

The wargames maintained by the Air Force Wargaming Center which have graphical interfaces have the east-west directions of the hexes across the points and the north-south directions across the flats. Since Saber is being created for the Wargaming Center's use, it was decided to change the direction of Ness' hexes to be the same as those used by the Wargaming Center. This permitted Klabunde to reuse code provided by the Wargaming

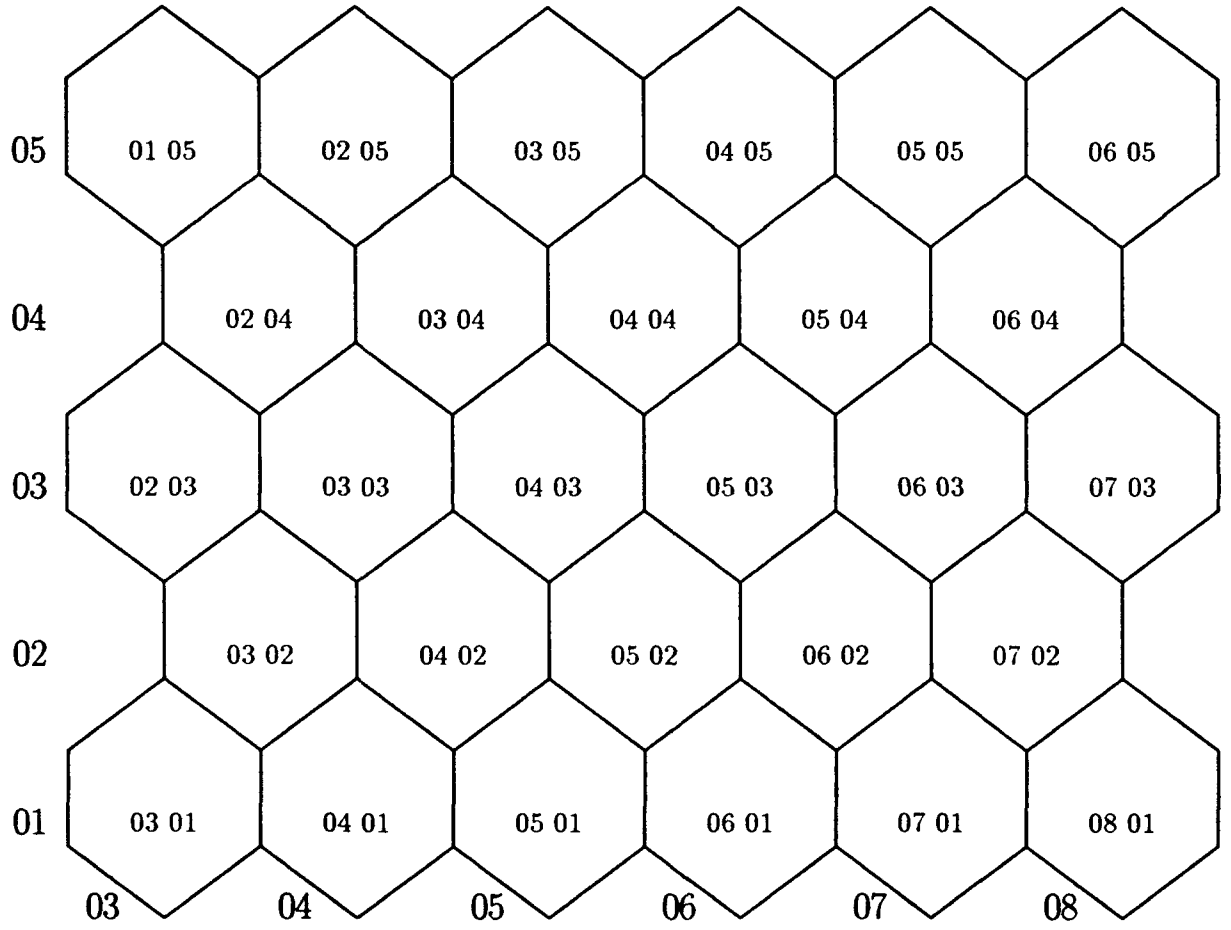


Figure 2. Ness' Hex Grid System

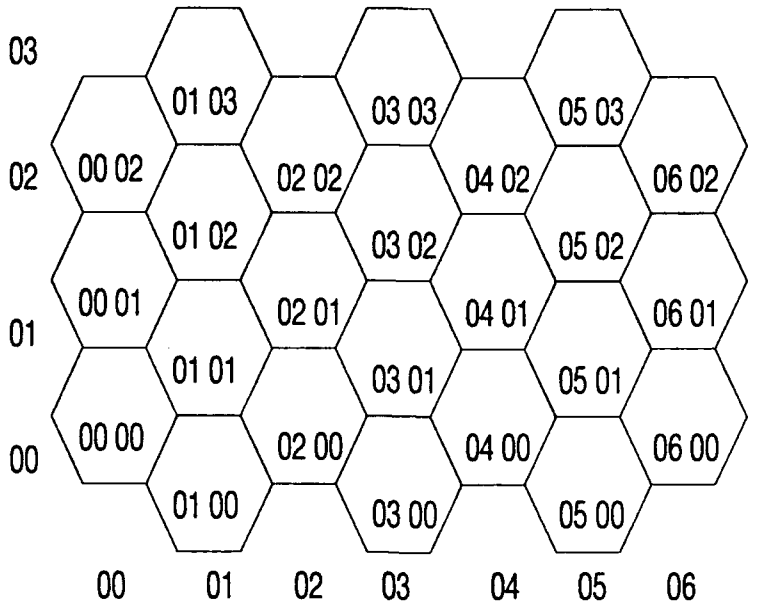


Figure 3. Revised Hex Grid System

Center. (13) Figure 3 shows the new direction of the hexes and the new numbering scheme. (11)

The addition of the air hexes meant an additional coordinate was needed to represent the altitude. This was done simply by adding a z-coordinate. Normally when one thinks of a three dimensional coordinate system, the z-coordinate is the third number. In the case of Saber, it was decided that it would be easier for the user to read the hex number if the altitude, or level, was the first number. Therefore, the hex numbers for Saber take the following format: zzxxyy. This constrains the number of hexes to two digits. Using base 10 numbers up to 100 × 100 hexes may be used for the ground level. At 25 km across, this implies a maximum playing area of 250,000 km^2. This will be sufficient for theater-level games, however, air assets located outside the playing area will need to be simulated somewhere on the allowed grid space.

It was also decided to reserve the minimum and maximum hex numbers for special use. They are used for two purposes. The first is in determining the correct movement

of units or aircraft packages. Having the extra hex numbers around the "edges" of the grid eliminates the need for a complicated movement algorithm that would have to check for those "edges". The second purpose is to use those same numbers to implement bases which are not within the map being depicted by the hex grid. For example, if the scenario being played is Korea and the USAF had a base in the Phillipines, that base might provide aircraft support to the forces in Korea. Some means was needed to represent this concept. In this example, the base might be assigned to ground hex 010000. The simulation will not attempt to fight a battle or drop munitions on the minimum and maximum hexes; i.e., they can not be targeted. They are basically placeholders and the resources located within them can "fly" out of, and return to, them. When an aircraft package leaves from one of these hexes, maintenance is determined from that location. No attempt should be made to determine maintenance from the "real" location.

4.2.2 Weather. The preprocessor provides the simulation with a good and fair weather forecast percentage. The forecasts are used in determining the actual weather for an individual hex. The weather for each hex should be determined by comparing the result of a random number draw with the forecast. If the random number is less than or equal to the forecast for good weather, then the actual weather is good. If the random number is greater than the forecast for good weather and less than or equal to the summation of the percentages for good and fair weather, then the actual weather is fair. The actual weather is poor if it exceeds the summation for the criteria for good or fair. The weather changes are based on weather periods and not on time periods. The weather period is a multiple of a time period. (24)

4.2.3 Asset ID. The database system design for Saber assigns an identifying number to each aircraft package, base, depot, ground unit, hex, obstacle, road, railroad, pipeline, Army order number, city, and satellite. This identifying number is referred to as an ASSET_ID. It consists of two letters and a six digit number. The letters used indicate what asset is being referred to. For example, a ground unit (or land unit) ASSET_ID begins with LU while a railroad begins with RR. The six digit number is a chronological number except in the case of a hex id. The first two numbers of a hex id indicate the altitude level (1-7, as described by Mann), the next two numbers refer to the x-coordinate of the hex while the last two numbers indicate the y-coordinate. (11)

Ness' code created a hex grid by using a loop and initializing attributes. The database design includes tables to provide this information. This includes not only the obstacle

values, but also the hex IDs themselves. In addition, the database design uses a neighbor ID to indicate the relationship between hex sides of adjoining hexes. For example, the NE side of one hex and the SW side of the adjoining hex would have the same neighbor ID. This allows the simulation to check the side of one hex for an obstacle and not have to check the side of the next hex to see if it has an obstacle as was required by Ness' design.

4.2.4 Staging Operations. Staging operations, including aircraft beddown, are performed by the preprocessor and not by the simulation.

4.2.5 Historical Data. Discussions pertaining to the postprocessor portion of Saber resulted in recognizing the need for a history file. This history file should be created by the simulation and is used in the processing of output reports. Entries to the history file should be made when certain events take place. Entries are of two types, events and status records. An event record contains information on things like getting attacked while status records log the current status of specific objects. For example, the events pertaining to an aircraft package are:

- Mission Start: This entry should include the mission start time, the asset ID, the rendezvous hex ID, the actual event, the mission type, the requested time-on-target, the force, and the target ID.
- Move: This entry records the movement of an aircraft package and consists of the following information: time, the asset ID, the new hex ID, and the event.
- Attacked By: This entry records the attack on an aircraft package by either another aircraft package or a ground unit (surface-to-air missile). It includes the following information: time, the asset ID of the aircraft package, the hex ID where the aircraft package is located, and the asset ID of the attacker. If aircraft package number AC000017 located in hex number 020120 was attacked at 1700 by aircraft package number AC000232, the event record would look like the following:

  E 1700 AC000017 HEX020120 ATKDBY AC000232.

- Jettison: This entry would indicate that weapons were jettisoned from an aircraft. It would enter information on the mission type, the aircraft type, and the number of weapons jettisoned.
- Mission Complete: The mission complete event record logs the fact that an aircraft package has completed its mission. It consists of the time of completion, the aircraft package number, the hex number, and the event.

The status records for an aircraft package would show the status of the aircraft package's aircraft after each of the events described previously. The status record would show the aircraft type and the quantity present in the aircraft package. Similar event and status entries should also be made for a base, depot, ground unit, satellite, supply train, ground level hex, and weather (see (14)).

4.2.6 Beginning and End-of-Day Routines. Input routines should only read in the data needed by the simulation. The routines should not read in attributes like full_designator which the simulation itself does not need.

At the end of each day two events should happen, output files should be written and output status reports should be created. Ness' code includes procedures to accomplish both of these events. These routines are no longer accurate. Klabunde has designed new reports (13). The procedures to create output files should merge the input files with the attributes which were modified by the simulation.

4.3 Land Module

4.3.1 Targets and Obstacles. The database design is flexible and permits a ground unit to have more than one target. It also permits a ground level hex to have more than one obstacle per side. (11) Ness' design was based on the ground unit having one target and a hex only having one obstacle per side.

The graphical interface design required the addition of pipelines, roads, railroads, and rivers. The addition of these items to Saber will permit them to be displayed. It should also permit them to be targeted. (13)

4.3.2 FEBA, Coasts, and Borders. Horton's database design includes tables which contain the location of the forward edge of the battle area (FEBA) as well as the location of coasts and borders. The coasts and borders are not needed for the simulation itself but rather are for graphical display purposes. Ness' land module already has a mechanism for determining the FEBA.

4.3.3 Ammunition, POL, and Firepower. The code for Mann's algorithm for determining the total POL (fuel) and AMMO (ammunition) for a ground unit should read in variable values from the database and not have any values hard-coded. The same is true for the decimals used in determining the BLUE and RED firepower scores. Mann shows,

for example, that the number of tanks is multiplied by .5. The .5 should be read in from the database and not be hard-coded. (17:176)

The presence of missiles does not increase a unit's firepower score because of the distant range of the missiles. Mann describes firepower and combat power differently from Ness. Mann's firepower is Ness' combat power and vice versa. The simulation should reflect Mann's definition. This includes Ness' attributes of CP_IN and CP_OUT.

4.3.4 Orders. The database design has a MOVE table and a MOVE_LNLT (MOVE_LEAVE NO LATER THAN) table for the Army orders. The MOVE table provides the input for the way Ness implemented orders and provides the leave no earlier than day and time period. The MOVE_LNLT table provides a new feature, that of overriding. It gives the user the ability to say that a unit must leave by a specific day and period.

It is used when a unit is delayed for some reason. The MOVE_LNLT orders have a higher priority for completion than the MOVE orders. For each time period within a day, the simulation should first determine whether there are any orders that were part of the MOVE_LNLT table that need to be processed. After it processes any orders of this type, it should then check for orders that were part of the MOVE table and are for that specific day and time period. (11)

4.3.5 Supply Train. A supply train moves supplies from a depot to either a ground unit or to a base. Since a supply train moves through ground hexes, it should have ground unit attributes and use ground unit algorithms. The supply train's additional attributes are needed to describe what supplies the train is moving and to what unit or base they should be delivered.

Saber should simulate two types of supply train missions, one in which the user would request each time needed (ST) and a predirect one which would run automatically. The predirect supply train (PST) would simulate the replenishing of supplies which is accomplished as a normal part of a base or ground unit's operation.

The supply train simulates movement by trucks. A supply train (ST) mission would deliver the supplies, leave the trucks, and disappear, unless the student submitted a second order to return the trucks to their depot. Therefore, the simulation must determine whether a supply train has a subsequent return order before adding the delivery trucks to the base or unit's inventory.

In contrast to the ST mission, one PST mission would deliver its supplies and return to its depot. The train would then be used for another PST mission. (24)

4.3.6 Radars. Mann describes an algorithm for determining the number of missiles fired. This algorithm requires the use of a probability that there is a launcher and radar pair (17:131). The type of radar that should be paired with a launcher is a radar fire control (RFC). Mann says that the launcher/radar pair should be determined by a random number draw. Instead, it was decided that the simulation should use the average radar quality. The following algorithm should be used to determine the radar quality:

Radar Quality = 1 / (maximum radar quality − actual radar quality + 1) (24)

The radar quality is calculated for each RFC type. The radar qualities are then summed and averaged. The average is used in place of the random number draw. The maximum radar quality is a user-supplied constant for each radar type. The actual radar quality should be a floating point number between 0 and 2. (24)

The same radar quality algorithm should also be used as part of Mann's local detection algorithm but instead of using RFC, the calculation should be based on acquisition radar. The revised local detection algorithm is as follows:

$$P(t) = (1/EC)(1 - e^{\frac{-(wWt)}{A}})(ARQ)$$

where P(t) is the probability of detection, EC is the electronic combat value of the target, w is the target speed, W is the diameter of the sensor's detection area, t is the time the target was in the area, A is the size of the search area (17:124) and ARQ is the average acquisition radar quality. (24)

4.3.7 Launchers. There is a one-to-one correspondence between launchers and missiles. There was some discussion about this correspondence because some launchers actually do have the capability of launching multiple missiles at one time. It was decided that it would be easier to implement ground missiles if the launcher could only fire one missile at a time. It is also easier to determine that the launcher is destroyed. If a launcher could fire three missiles simultaneously, then what would be the result of a hit from an aircraft? Would it destroy the ability to fire one missile, two missiles, or all three? Was the launcher fully loaded or partially loaded when it was hit? The one-to-one correspondence makes the implementation easier but still realistic.

4.3.8 Surface Missiles. Since surface missiles are attached to a ground unit, they need to have the attributes of a ground unit. But since they are also missiles, they need additional attributes. Therefore, when simulating a surface-to-surface missile mission, the simulation must use the attributes of both a ground unit and a surface-to-surface missile. Surface-to-air missiles do not need a separate mission because they only fire in response to an aircraft package's presence. The processing of an aircraft package will check for surface-to-air missiles and, if any are present, will determine the results.

Surface missiles need to have attributes for the probability of kill (PK) for a hard, soft, and medium target. These attributes are needed so that the simulation can determine the outcome of a conflict. Each of the targets has a hardness. This hardness value provides the simulation with the ability to provide different results based on the type of target. For example, the results of firing a missile at an aircraft should not be the same as firing one against a runway. (24)

4.4 Air Module

4.4.1 Air Hexes. The following subsections describe the clarifications, modifications, and enhancements to the air hexes.

4.4.1.1 Relationship with Ground Hex. A method was needed to relate the ground hexes to an air hex. Seven ground hexes were grouped to create an air hex. Figure 4 shows the relationship between a ground hex and an air hex (11). It was determined that one way of relating the ground hexes to the appropriate air hex was to give the ground hex an additional attribute of CENTER_HEX. This value is the hex id of the center ground hex when grouped by sevens. It is also the number used for the corresponding air hex. Figure 5 shows how seven ground hexes are related to an air hex via the CENTER_HEX number (11).

4.4.1.2 Trafficability. The movement of aircraft packages through an air hex should not only be based on weather and the electronic combat (EC) value of the hex, as described by Mann, it also needs to be based on trafficability (17:61). Mann describes the second hex level as being tree-top level; therefore, some method of simulating the trees is needed. Mountains could also be located in an air hex. The addition of an attribute for trafficability needs to be added. It should be implemented similar to the way the land module uses it.

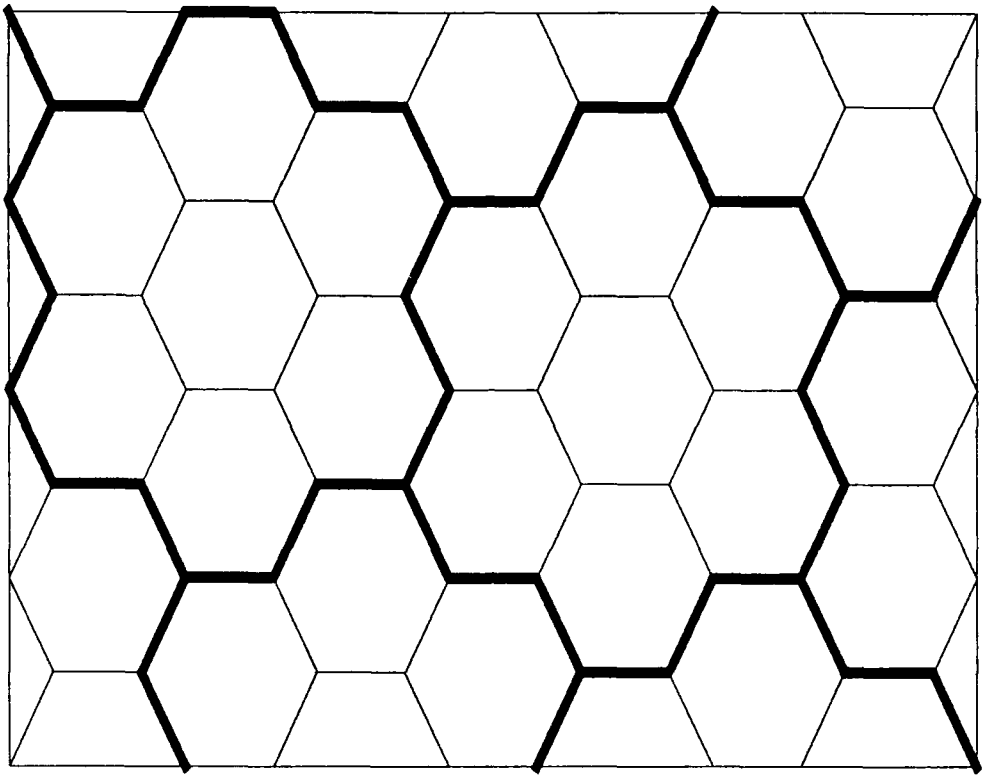


Figure 4. Air Hex Grid

4.4.1.3 Aircraft Package Movement. Mann said an aircraft package should move through air hexes by determining a straight line from the rendezvous hex to the target or destination hex (17:119). Many discussions took place on how this should be accomplished. It was decided that an aircraft package should move from hex to hex and resolve any conflicts in each hex before proceeding. It was also decided to determine the route by moving through each hex and calculating the next hex location based on the current hex location. The six directions which an aircraft package can move from hex to hex are north, south, north-east, north-west, south-east, and south-west; an aircraft package can not move directly east or west.

It was decided that if a package was moving south it would first move down and left. On the next move it would go down and right. This rotation of left and right would continue until the aircraft package reached its destination hex. For example, if the start hex number was 020516 and the destination hex was 020502, the aircraft package would first move to 020413, the next moves would be to hexes 020611, 020509, 020406, and finally

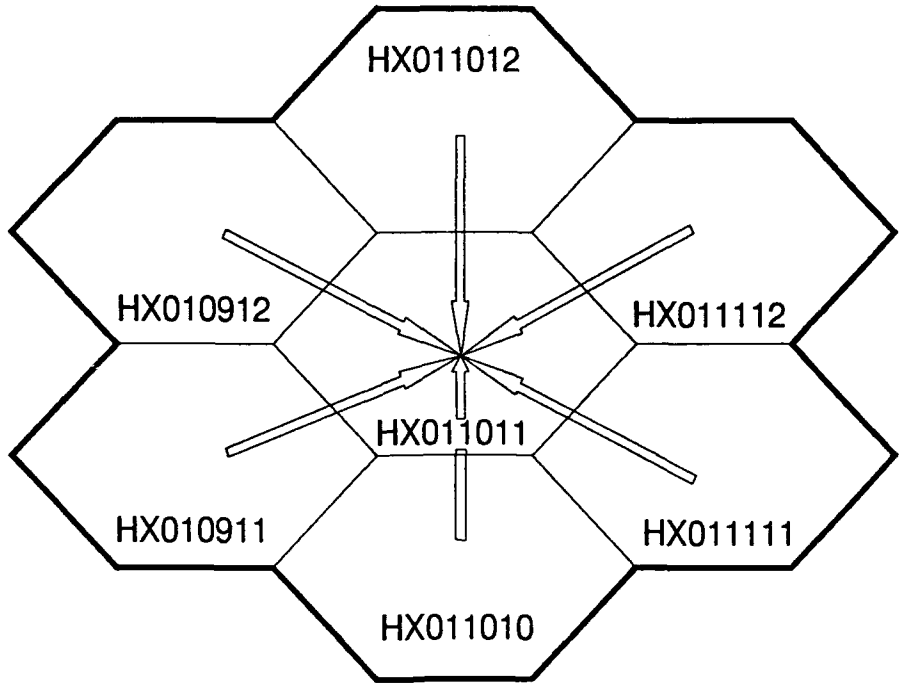


Figure 5. CENTER_HEX Concept

to hex 020502. The opposite is true if the package is moving north. To go from 020502 to hex 020516 the aircraft package would move first to 020604, then to 020707, 020509, 020611, 020714 and finally to 020809. Figure 6 shows the north-south and south-north movement.

If the aircraft package wants to move in a northeasterly direction from its starting location to its destination location it first moves up and to the right and then up right. For example, to move from air hex 020502 to air hex 022414, the aircraft package would first move to hex 020802, then to hex 020905. From there it goes to hexes 021205, 021308, 021608, 021711, 022011, 022114, and finally reaches its destination of 022414. Going southwest, the algorithm does the opposite, the simulation moves down and to the left and then down left. In this example, the aircraft package would go from 022414, to 022114, 022011, 021711, 021608, 021308, 021205, 020905, 020802, and finally to air hex 020502. Note, this does take the aircraft package through the same air hexes as the northeast

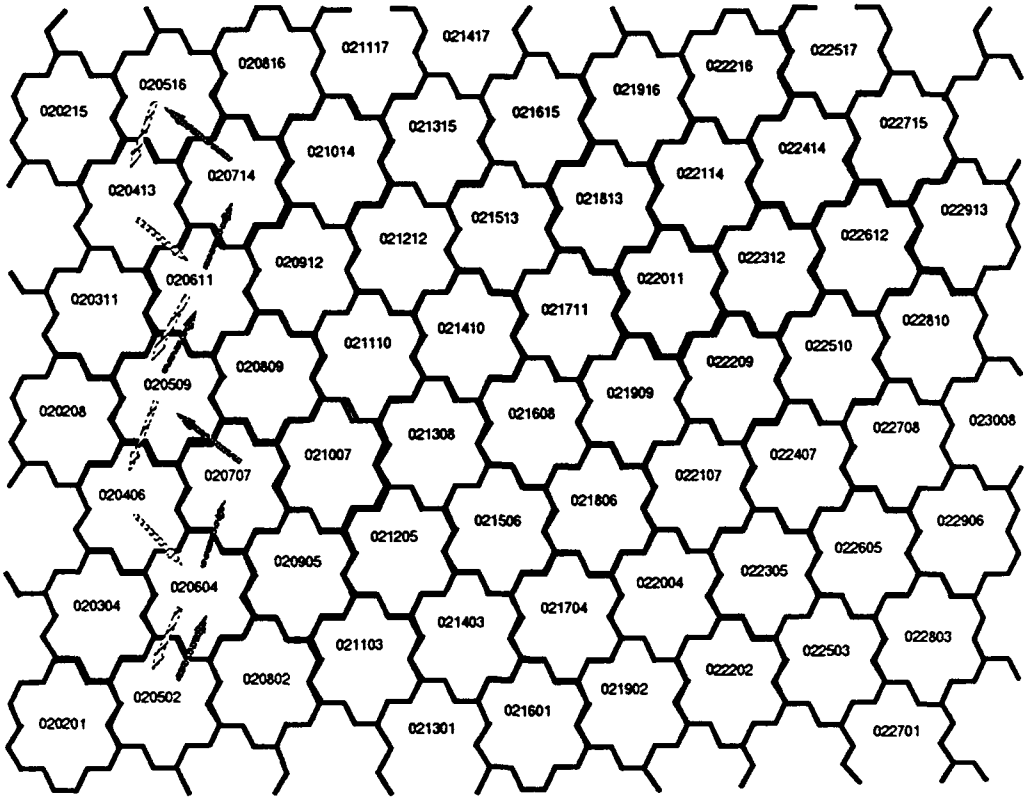


Figure 6. North-South and South-North Movement

direction. Figure 7 shows the northeast and southwest movement of an aircraft package through air hexes located in level 2.

If the aircraft package wants to move in a southeasterly direction it first moves up and to the right and then moves down to the right, alternating directions until it reaches its final destination. For example, an aircraft package moving from air hex 020413 to 022503 would first move to hex 020611 then to hexes 020809, 021007, 021205, 021403, 021601, 021902, 022202, and finally to 022503. For a northwest movement, the aircraft package would go from 022503 to 022305, 022107, 021909, 021711, 021513, 021212, 020912, 020611, and then reach its destination of 020413. It attempts to first go up left and then up right. Again, the simulation takes care of those situations when the aircraft package cannot alternate directions. Figure 8 shows the southeast and northwest movement of an aircraft package through air hexes located in level 2.

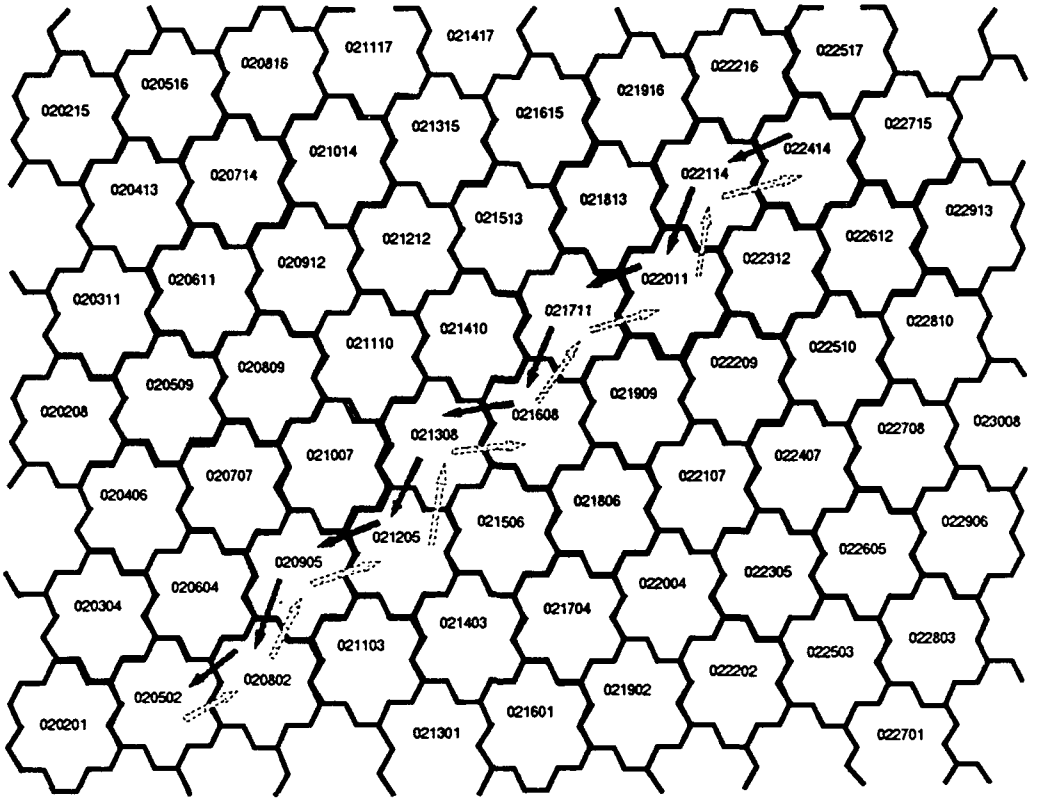


Figure 7. Northeast and Southwest Movement

4.4.2 Weapons Load. There are three types of weapons load: preferred conventional load (PCL), preferred nuclear load (PNL), and preferred biological, or chemical, load (PBL). When the user enters the missions to be flown, the type of warhead to be used should also be entered.

When forming an aircraft package, the system should first determine if the requested aircraft are available. After it determines that the required aircraft are available, it should then determine the actual weapons to use. The particular weapons load needed is first based on whether the mission is using conventional, nuclear, or chemical warheads.

A particular weapons load is based not only on the warhead but also on the aircraft being used, the weather of the target hex, the hardness of the target, and the mission to be flown. For example, an F15A flying a Suppression of Enemy Air Defense (SEAD) mission with conventional weapons in poor weather against a hard target would require a different make-up of weapons than the same aircraft flying a fighter sweep mission. (24)

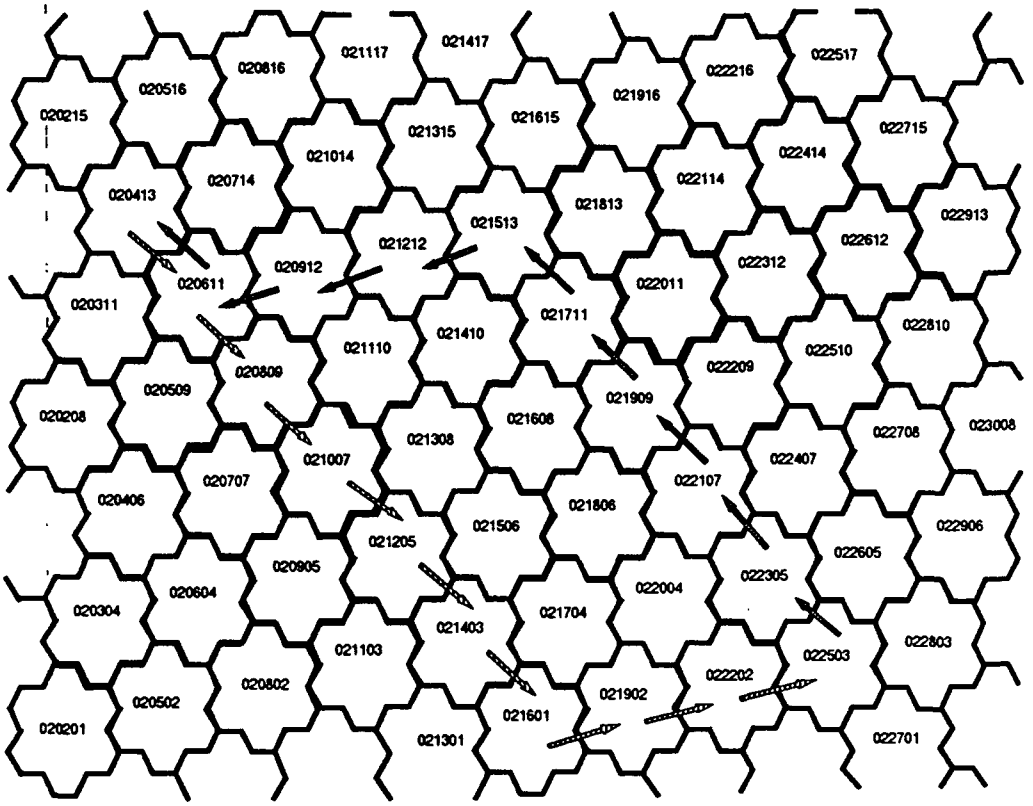


Figure 8. Southeast and Northwest Movement

The names of the weapons must be unique even though the same weapon might carry a conventional, nuclear, or chemical warhead. For example, a Patriot missile can have either a nuclear or conventional warhead. The Patriot's designation is MIM-104. (31) For Saber, the Patriot's designation would be MIM-104C when it has a conventional warhead and MIM-104N when it has a nuclear warhead.

4.4.3 Aircraft Maintenance. Aircraft require maintenance each time they are flown on a mission. Each aircraft type has an attribute of maintenance distribution which refers to the type of statistical distribution that should be used by the simulation to calculate the actual maintenance time. An aircraft record also has attributes for the mean and standard deviation. The types of distributions initially allowed are: normal, poisson, and uniform. (24)

4.4.4 Base Missions. Mann says a base should have three missions: NA, fire, and move. The fire mission was deleted from the requirement because fire support would be

received from a ground unit. The move mission was renamed deploy. Bases don't really get up and move like ground units but personnel and aircraft are often deployed. Therefore, the name was changed. The deploy mission would permit an entire base to move from one hex location to another. (24)

4.4.5 Aircraft Packages. In addition to an aircraft package's primary mission, an aircraft package can also have escort, SEAD, ECM (Electronic Counter Measures), and refueling aircraft assigned to it. If an aircraft package encounters an enemy aircraft package, the escort and SEAD planes are targeted first. The presence of SEAD and ECM aircraft increase the electronic combat (EC) rating for the aircraft package. They can also increase the distance that the aircraft package can fly. An aircraft package would continue on its mission even if the escort aircraft are destroyed. SEAD aircraft can be either a support aircraft or perform its own mission. When used as a support aircraft it strikes enemy air defense at the target.

An aircraft package is formed by taking aircraft from various bases. Before selecting an aircraft for a mission, the simulation must ensure the following:

- the runway length is sufficient for take-off,
- the aircraft is not currently undergoing maintenance,
- the base has enough fuel available for the mission,
- the correct weapons load is available,
- the aircraft can fly in the target hex's current weather,
- the target is within the range of the aircraft or refueling planes are available.

Fuel and weapon quantities should be reduced when the aircraft package is formed. Spare parts and maintenance hours are decreased when the aircraft returns from a mission. If the aircraft package was hit, then the aircraft would require more than routine preventive maintenance, i.e., the maintenance time needs to be longer.

Once the simulation gathers the number of aircraft needed, the aircraft automatically appear at the rendezvous hex. The simulation does not need to "fly" the planes to the mission's starting point. When an aircraft package returns from a mission, the aircraft are randomly distributed back to the bases.

4.4.6 AF Missions. Any aircraft can fly any mission. The aircraft's COMBAT attribute provides a combat rating which ensures that a F16 flying an escort mission has better results than a KC135 flying the same mission. The preprocessor will simulate warnings to the player of unreasonable missions.

Each mission can only target one hex. Most missions target a base or unit. A reconnaissance (RECCE) mission is different in that it can have a hex id as its target. Once the RECCE plane reaches the hex it determines what bases, supply trains, or units are in the hex. Anything which it finds in the hex that can be a target becomes a target. The simulation then determines the order used in striking the targets. Both chemical and nuclear missions can also target a hex. If a Battlefield Air Interdiction (BAI) or Air Interdiction (AI) mission has a hex ID as its target, then a mine laying operation should be performed.

4.5 Summary

This chapter described changes and enhancements to Saber based on research being conducted by Klabunde and Horton. It also clarified some of Mann's algorithms. Specifically, the chapter described Ness' hex system and how the hex system was changed to provide compatibility with other wargames used at the Wargaming Center and to provide a better visual display of the hex grid. This chapter also discussed the changes in how the actual weather is determined. The new method is stochastic in design.

Each asset in the wargame has a unique identifying number called an ASSET_ID. Aircraft and weapons do not have ASSET_IDs because the simulation is not tracking tail numbers of aircraft or individual weapons; rather the simulation just keeps track of quantities of different aircraft and weapons.

Since there is a graphical interface to Saber and since the interface is not real-time, a history file is needed to store what actions the simulation has taken. This requires additional code in the land battle module. Other changes to the land module include the capability to have more than one obstacle in a hex and for a unit to be able to target more than one other unit. In addition, there now exists the capability to target railroads, roads, pipelines, and rivers.

A ground unit can have two kinds of orders with Saber. Ness' design used a MOVE table; Horton's adds a MOVE_LNLT table. Supply trains can be of two types, predirect, which runs automatically, and one the student requests. Both supply trains and surface missiles have the attributes of a ground unit in addition to their own attributes.

This chapter also described clarifications and modifications to the requirements of the air module. An air hex, like a ground hex, can have trafficability. In an air hex, this attribute indicates the presence, or absence, of mountains and tall trees.

Aircraft can have three types of weapons loads: preferred conventional load, preferred nuclear load, and preferred chemical/biological load. This chapter described the process of choosing the correct weapon load based on the mission, the target hardness, and the weather. This chapter also described how aircraft maintenance should be calculated.

Aircraft packages have a primary mission with primary aircraft plus support aircraft for escorts, refueling, suppression of enemy air defenses, and to provide electronic counter measure operations.

The support aircraft are not a required part of an aircraft package but provide additional capabilities if included. The formation of an aircraft package was described. It does not include ensuring that required maintenance hours and spare parts are available. These are changed when the mission is completed.

The next chapter will describe the detailed object-oriented design.

V. Detailed Object-Oriented Design

5.1 Introduction

This chapter describes the results of the detailed object-oriented design. This phase consisted of finalizing the objects, their attributes, the operations needed, establishing the visibility between objects, and establishing the interface of each object.

The first four of Booch's object-oriented design steps, as discussed in Chapter 2, were used to refine the initial object-oriented design and determine the final set of objects. This chapter is divided into sections based on Booch's design steps.

5.2 Identify the Objects and Their Attributes

As stated in Chapter 3, the first step in this process is to determine the objects. This was accomplished by extracting nouns from the requirements documents (Ness and Mann's theses) and by discussing the model design with the other wargaming thesis students.

The final list of objects and their attributes from the initial object-oriented design were refined as part of this step. The final objects and their corresponding attributes are shown in Appendix E. The items shown in italics were not part of the initial design. In Ada, objects are encapsulated in packages. The following sections describe the various packages.

5.2.1 AFSim and ArmySim. AFSim and ArmySim are two separate packages but perform similar functions. Neither contain any object declarations. They are the top-level simulation objects for Saber. AFSim is used to simulate air activities and ArmySim is used to simulate land activities. Both packages call procedures and functions from other packages. The idea for these objects came from a MITRE working paper which used an object called The Simulator as its top-level object. (6:24)

5.2.2 Aircraft. This package contains the declarations for an aircraft. It includes the declarations for the three different types of weapons loads: preferred conventional, preferred nuclear, and preferred biological (chemical). A weapons load is based on the type of aircraft, the mission, the target hardness, and the weather. A consideration was made as to whether each of the loads should by themselves be objects but it was decided to locate the description within the Aircraft object. The reason for this is because once the initial weapons load information is read, it is not changed.

This package also includes the declaration for AF_MISSIONS. This declaration is an enumeration type of various air force missions, such as BAI (battlefield air interdiction), AI (air interdiction), CAS (close air support), etc. Though a number of objects also have an attribute of AF_MISSIONS, this is the first location where it is needed. As seen later in this chapter when visibility is discussed, all the other objects which need AF_MISSIONS also need the AIRCRAFT object.

5.2.3 AircraftPackage. The AircraftPackage object consists of all the declarations needed to define an aircraft package. It also includes those user-defined constants which pertain directly to an aircraft package.

5.2.4 AirHex, GroundHex, and Hex. The initial design had an object called Hex which was to represent the terrain features. After additional discussions and further design work, it was determined that two objects would be needed, one for the air hexes and a second for the ground hexes. This is because the air hexes do not need most of the attributes of the land hex.

A third object was created later and contains declarations which are common to both the land and air hexes, as well as declarations for some of the ground unit attributes which were also needed by the ground hex object. Weather and trafficability are examples of attributes which are common to both hexes.

The GroundHex package includes declarations for things like roads, pipelines, and railroads. These attributes were added to Ness' GroundHex declaration because they should be used in the simulation when determining trafficability. They should also be used for logistics and as targets. For example, a railroad runs along the NE side of hex number 013323 and a unit wants to move into that hex. The presence of the railroad should make the movement through the hex more difficult to represent the presence of a train running on the tracks. If a road ran through the same hex, the presence would make trafficability easier but if the road was destroyed it should be more difficult for the troops to move through the hex. This is simulated by decreasing the trafficability from, for example, excellent to poor. The roads, pipelines, and railroads should also be used to provide logistical support to other land units and bases. The movement of a supply train can be simulated by routing it via the roads or railroads. Pipelines can be used to simulate the resupply of fuel or water to a unit or base. Destroying a portion of a pipeline would cut off the fuel or water supply. Again, destroying a portion of a road or railroad would make the supply train's movement more difficult (or impossible without changing routes).

5.2.5 Algorithms. This object contains no variable declarations but is used to encapsulate the mathematical operations needed for the simulation. Further discussion of this object is deferred until the section describing operations on an object.

5.2.6 Bases. The Bases object includes not only the declaration for an airbase but also a Depot. The difference between the two is how they are used. An aircraft package is formed by taking resources (for example, aircraft, weapons, POL, spare parts) from a base and using them in support of a mission. A supply train is formed by taking resources (such as weapons, POL and spare parts) from a depot and delivering them to a base or ground unit. Though the USAF uses aircraft to resupply bases, the simulation will not do so. Resupply by aircraft will be done by the preprocessor. A depot, including the aircraft, should still be a target for the simulation, hence the need for the object.

5.2.7 Clock. This object contains the constants needed by the simulation to keep track of the day being processed as well as the constants used by the simulation for the number of days and the number of periods.

5.2.8 Forces. This object was also not a part of the initial design. It was added because the Aircraft, Weapons, GroundUnits, Bases, Depots, and Missiles all contain an attribute of Forces. It consists of two declarations: FORCES_TYPE which can be either BLUE, RED, or NEUTRAL and COUNTRY_TYPE which links particular countries to a specific force or side. Having a separate package for this declaration will make it easy to enhance the FORCES_TYPE to include more than the current two forces — BLUE and RED.

5.2.9 GroundUnits. This object came directly from Ness' work and is the basic operational object for the land battle that the players manipulate (20:47). The specification for this package includes all the user-defined variables which are needed for the land simulation and pertain to a ground unit. It also includes variable declarations needed to define the attributes of the ground unit. For example, one of the attributes of an Army unit is TYPE_OF_UNIT. Therefore, this package contains Ness' declaration for ARMY_UNITS.

This package also contains the declaration for a supply train since a supply train uses the attributes of a GroundUnit and also the ground unit algorithms. A supply train also has additional attributes.

5.2.10 Radars. This object was added to the list of objects based on further discussions of the need for, and implementation of, ground missiles. The Radars package contains the declaration for a radar. Radars can be of two types, radar fire control and acquisition (17:102).

5.2.11 OneWayLists. This object was added. It defines a generic one-way linked list. Except for the addition of the Ada code needed to create a generic package, it was taken directly from Feldman (9:110). With this generic object, an object can be implemented as a list without requiring the type declaration and operations to be defined within each package.

5.2.12 Satellites. The need for this object was not seen initially. That is because it was thought of as part of AF_MISSIONS. In order to conduct the satellite mission an object is needed. The attributes for a satellite are taken directly from Horton's database design.

5.2.13 TargetInfo. This is another object which was not part of the initial design. This package contains the declarations for the hardness of the target and the declarations for the various components which make up a ground unit or a base and are, hence, targetable. These components include, for example, the different types of tanks and trucks. This package also includes the declaration for cargo and provides the capacity of a cargo unit. The need for these items is based on Mann and is used in determining the results of an air strike (17:150).

5.2.14 UniformPackage. This object is a generic package. This object provides the ability to determine a uniform random number. It contains only generic type declarations for a range of floating point numbers. This permits the user to define the ranges needed for specific applications.

5.2.15 Weapons. This package contains the declarations for the air-to-air and air-to-surface weapons as well as the surface-to-surface and surface-to-air missiles. It provides the ability for a weapon to contain either a nuclear or a biological/chemical warhead.

5.3 Identify the Operations Suffered By and Required of Each Object

As stated in Chapter 3, the operations are determined by extracting verbs and verb phrases from the requirements documents. Operations are the various procedures and

functions which are needed by the simulation to manipulate the object. The final version of the objects and their required operations are shown in Appendix F. The items in italics were not part of the initial design. Below is a brief description of the operations needed for a specific object. The procedures and functions described are those that need to interface with other objects. Additional procedures and functions are sometimes required by the ones discussed below. Those additional procedures and functions will be discussed with the implementation.

5.3.1 AFSim. This package contains all the procedures needed to run the air portion of the simulation.

- INITIALIZE. This procedure should initialize all the objects related to the air simulation.
- PERFORM_MISSIONS. This procedure should determine what mission needs to be performed and call the appropriate procedure to accomplish that mission.
- RUN_DELAYED_MISSIONS. This procedure is needed to run any missions that were delayed from the previous time period.
- WRITE_DATA. This procedure should provide the capability to write all the data back to disk.
- REPORT_WRITER. This procedure should provide the ability to print required reports.

5.3.2 Aircraft. This section describes the operations needed by the Aircraft object.

- GET_AC. This procedure should read in from the disk file the initial aircraft information.
- INCREASE_AC. This procedure provides the ability to increase the number of aircraft. It should be used to increase aircraft assigned to particular bases when breaking up an aircraft package upon the completion of a mission.
- DECREASE_AC. This procedure provides the ability to decrease the number of aircraft. It should be used to decrease aircraft from the appropriate bases when forming an aircraft package. It should also be used to decrease the number of aircraft that are part of an aircraft package to reflect losses due to combat.

- ADD_TO_MAINT_LIST. Once an aircraft is flown on a mission it must undergo maintenance. This procedure should provide that capability.
- DELETE_FROM_MAINT_LIST. Once an aircraft has received its maintenance it should be made available for new missions. This procedure should provide this capability.
- DETERMINE_PCL. This procedure determines the preferred conventional weapons load which a specific aircraft should use on a specific mission.
- DETERMINE_PBL. This procedure determines the preferred biological/chemical weapons load which a specific aircraft should use on a specific mission.
- DETERMINE_PNL. This procedure determines the preferred nuclear weapons load which a specific aircraft should use on a specific mission.
- GET_AC_QUANT. This procedure should read in from disk the quantities available for each type of aircraft.
- GET_REFUEL_CAP. This procedure should read in from disk those aircraft which can be refueled.
- GET_MAINT. This procedure should read in from disk the maintenance information for each type of aircraft.
- WRITE_AC_QUANT. This procedure should write out to disk the aircraft quantities.

5.3.3 AircraftPackage. This section contains descriptions of the procedures needed by an AircraftPackage object.

- GET_ACPKG_INFO. This procedure should read in from the disk file the information needed for the various aircraft missions.
- FORM_ACPKG. This procedure should form the aircraft packages needed to accomplish the various missions read in by the previous procedure.
- BREAKUP_ACPKG. This procedure should return any resources not used by the aircraft package back to the appropriate bases. It should also ensure that the returning aircraft undergo maintenance.
- DELAY_MISSION. This procedure should add a delayed mission to a list of delayed missions.

- GET_CONSTANT. This procedure should read in the constant required by the AircraftPackage object in order to fine-tune the simulation.
- DECREASE_ASSETS. This procedure should decrease the resources held by an aircraft package based on the resolution of any conflicts.
- DELETE_ACPKG_MISSION_LIST. This procedure should delete an aircraft package from the list of missions once the package is formed.
- DELETE_ACPKG_DELAY_LIST. This procedure should delete an aircraft package from the list of delayed missions once the package is formed.
- UPDATE procedures. There are a number of procedures needed by the simulation to update specific attributes of an aircraft package. See Appendix F for the names of all the procedures. Each procedure has the same basic function: it changes the value in the attribute which matches the name of the procedure. For example, the UPDATE_INTEL_INDEX should change the INTEL_INDEX attribute of a specific AircraftPackage. Having the simulation (in AirSim) determine what the actual value is and then just replacing the value with an UPDATE procedure reduces the number of procedures needed since there is no need for separate INCREASE and DECREASE procedures. The same concept is used for all the other objects.
- WRITE_ACPKG_INFO. This procedure should write out to disk the information on the various aircraft package missions.
- WRITE_DELAY_MISSIONS. This procedure should write out to disk the information on delayed missions so that they can be read back in from disk for the next day.

5.3.4 AirHex. This section describes the procedures needed for the AirHex object.

- GET_AIR_GRID. This procedure should read in the initial air hex data from the input disk file and create an AIR_GRID.
- LOAD_WEATHER. This procedure loads the actual weather into each air hex.
- ADD_ACPKG. This procedure adds an aircraft package to a specific air hex. This is especially needed by the area missions which remain in an air hex for at least one time period.
- DELETE_ACPKG. This procedure removes an aircraft package from an air hex once the package is either destroyed in its entirety or once it moves to another air hex.

- ADD_SATELLITE. This procedure should add a satellite to an air hex.
- DELETE_SATELLITE. This procedure should delete a satellite from an air hex.
- UPDATE_LOCATION. This procedure should update the location of a specific aircraft package.
- WRITE_AIR_GRID. This procedure should write out to disk the air hex grid information.

5.3.5 Algorithms. The Algorithm object contains procedures and functions for all the new mathematical calculations required by the simulation. It does not include those calculations which Ness created as part of his land battle simulation. The idea for this object came from a MITRE working paper. The Battefield Environment Model, which the paper describes, contains not only the object, The Simulator, described earlier, but also one called the Mathematician. Their Mathematician object performs most of the simulation calculations. (6:31) The Algorithms object should also perform most of Saber's calculations as described by Mann's Chapter 6. The procedures and functions required are described below.

- DETERMINE_DIRECTION. This procedure determines the direction in which an aircraft package should move.
- CALC_LOCAL_DETECTION. This procedure determines the local detection.
- ONE_MISS_ONE_AC. This function returns the probability that the missile successfully hits the aircraft.
- MULT_MISS_ONE_AC. This function returns the probability that the multiple missiles successfully destroy the aircraft.
- MULT_MISS_MULT_AC. This function returns the aggregated probability of kill based on multiple missiles and multiple aircraft.
- CALC_NO_MISS_FIRED. This procedure determines the number of missiles fired.
- CALC_BERNOULLI. This function performs a bernoulli random number draw based on the probability of kill. The result is used to determine attrition.
- CALC_BINOMIAL. This function performs a binomial random number draw and is based on the probability of kill. The result is used to determine attrition.

- CALC_POISSON. This function determines the poisson distribution and is used to determine the maintenance hours needed. This procedure is not based on Mann's thesis.
- CALC_NORMAL. This function determines the maintenance hours needed for an aircraft based on a normal distribution. This procedure is not based on Mann's thesis.
- CALC_PROB_HIT_CIRC_TARGET. This function determines the probability of hitting a circular target.
- CALC_PROB_HIT_RECT_TARGET. This function determines the probability of hitting a rectangular target.
- CALC_BLUE_FIREPOWER. This function determines the firepower value for a BLUE ground unit.
- CALC_RED_FIREPOWER. This function determines the firepower value for a RED ground unit. This function is not the same as the one for the BLUE side.
- CALC_POL. This function is used to calculate the total POL for a ground unit.
- CALC_AMMO. This function is used to calculate the total amount of ammunition a ground unit has.
- CALC_PROB_LAUNCH. This function calculates the probability of launching a missile while conducting air-to-air combat. This function should be used to calculate the probability of launch for both the RED and BLUE forces.
- CALC_SSPK. This function calculates the single shot probability of kill and is used for air-to-air combat.
- CALC_PROB_KILL. This function calculates the probability of kill for a missile fired from an aircraft and targeting an opposing aircraft. Like the CALC_PROB_LAUNCH function, this function should also be used for both RED and BLUE forces.
- DETERMINE_PROB_FOR_TARGETS. This function should determine the probability of an aircraft hitting a specific ground target.
- CALC_ADA_PK. This function should determine the probability of kill based on the air defense artillery values.
- DETERMINE_LONG_RANGE_DETECTION. This procedure should determine the long range detection when an aircraft package enters a new air hex.

5.3.6 ArmySim. The procedures which are part of the ArmySim package were designed and written by Ness. Ness did not have them encapsulated in one package; instead he used the Ada "separate" statement. They were encapsulated into one package in an effort to keep the design object-oriented. For the sake of completeness, the procedures are described in this section.

- INITIALIZE. This procedure initializes the units and ground hexes. It also sets up the initial pointers between the hexes and the units.
- REPORT_WRITER. This procedure creates output reports.
- LOG_SPT. This procedure controls the logistics operational flow of control for the ground battle.
- MOVEMENT. This procedure controls the movement of units.
- INTELLIGENCE. This procedure performs Army intelligence operations.
- WRITE_DATA. This procedure writes the hex and unit information to a disk file.
- SET_UP. This procedure performs the necessary combat set up operations prior to performing the attrition operations.
- ATTRITION. This procedure performs the overall control for combat attrition operations.
- PERFORM_ST_OPS. This procedure was not part of Ness' thesis work. It should perform supply train operations.

5.3.7 Bases. This section describes the operations required by, and needed for, an airbase or a depot.

- GET_BASE. This procedure should read in from a disk file the starting information for all the bases to be used in the simulation.
- GET_DEPOT. This procedure should read in from disk the initial information on depots.
- INCREASE_RESOURCES. This procedure should increase the aircraft and weapons on a specific base. It should be used when breaking up an aircraft package.
- DECREASE_RESOURCES. This procedure should decrease the aircraft and weapons available on a specific base. It should be used when forming an aircraft package or a supply train.

- INCREASE_SUPPLIES. This procedure should increase the amount of POL, spare parts, and weapons. It should be used as the result of a supply train operation.
- DECREASE_MAINT_NOS. This procedure should be used to decrease the MAINT_PERS_ON_HAND, MAINT_HRS_ACCUM, and MAINT_EQUIP_ON_HAND attributes of a base. This should be used when an aircraft returns from its required maintenance period as a result of conducting a mission.
- UPDATE procedures. There are numerous procedures needed to update the different attributes of a base or depot. These all begin with UPDATE and are followed by the name of the attribute which they are modifying. See Appendix F for the individual procedure names.
- WRITE_BASES. This procedure should write out to disk the information on bases.
- WRITE_DEPOTS. This procedure should write out to disk the information on depots.

5.3.8 Clock. The Clock package has two operations: GET_CONSTANTS which will read in the Clock related constants needed to fine-tune the simulation and GET_CYCLES which reads in from disk the cycle or period numbers needed by the simulation and a field telling whether the cycle is considered daytime or nighttime.

5.3.9 Forces. This package contains one procedure: GET_COUNTRIES. This procedure reads in from a disk file the country and its corresponding side (or force).

5.3.10 GroundHex. This package contains the procedures and functions needed for the object GroundHex. It includes many procedures written by Ness.

- GET_GRID. This function was part of Ness' original code and created a grid of hexes.
- GET_CONSTANTS. This procedure should read in all the user-defined constants needed to fine-tune the part of the simulation which relates to the ground hexes.
- CALC_VAL. This function was part of Ness' implementation. It assigns a value to the six possible difficulties used for obstacles.
- INITIAL_PTR. This procedure was part of Ness' implementation and initializes the hex-to-unit pointers.
- ADD. This procedure was part of Ness' implementation and adds a unit to a specific hex.

- DELETE. This procedure was part of Ness' implementation and deletes a unit from a specific hex.
- ADD_BASE. This procedure should add an airbase or depot to the list of bases located within a specific ground hex.
- DELETE_BASE. This procedure should delete an airbase or depot from the list of bases located within a specific ground hex.
- LOAD_WEATHER. This procedure should load the weather into each hex.
- ASSESS_CONTACT. This procedure was part of Ness' implementation. It initializes the contact parameters for each hex and unit, determines if opposing forces are in adjacent hexes, and sets the appropriate contact and attrition flags.
- APPLY_FS. This procedure was part of Ness' implementation. It applies field artillery, aviation, and air defense fire support with the combat power of the units being provided the support.
- APPLY_FP. This procedure was part of Ness' implementation. It sets up the numbers required for calculating attrition. It takes all the firepower in a hex and applies it equally to all adjacent hexes which contain opposing forces.
- ATTRIT. This procedure was part of Ness' implementation. It performs the attrition on every unit in combat.
- SF_INTEL. This procedure was part of Ness' implementation. It performs special forces intelligence operations.
- UPDATE procedures. There are numerous procedures needed to update the different attributes of a ground hex. These all begin with UPDATE and are followed by the name of the attribute which they are modifying. See Appendix F for the individual procedure names.
- WRITE_GRID. This procedure should write out to disk the ground hex information.

5.3.11 GroundUnits. This package contains the procedures and functions needed for the GroundUnits object.

- GET_CONSTANTS. This procedure reads in from disk the user-supplied constants needed to fine-tune the ground units.
- GET_UNIT. This procedure was initially written by Ness and reads in the ground unit information from a disk file.

- GET_SF_INTEL. This function reads in from disk the initial special forces information.
- GET_SUPPLY_TRAIN. This procedure should read in from disk information on the various supply trains.
- ADDIT. This procedure was part of Ness' implementation and adds a destroyed unit to a list of destroyed units.
- UNIT_LOG_SUPPLY. This procedure was part of Ness' implementation and adds supplies to appropriate units.
- REDUCE_INTEL. This procedure was part of Ness' implementation. It performs a reduction of intelligence for each time period to reflect a degradation of intelligence over time.
- UNIT_VAL. This function was part of Ness' implementation. It assigns a movement factor for combat units.
- MOVE_IN_GRID. This procedure was part of Ness' implementation. It performs the operations necessary when a unit moves within a hex.
- DELETE_SUPPLY_TRAIN. This procedure should delete a specific supply train from a specific ground hex.
- INCREASE_ST_ASSETS. This procedure should increase the POL, ammunition, hardware, and spares which the supply train will move from a depot to a ground unit.
- DECREASE_ST_ASSETS. This procedure should decrease the POL, ammunition, hardware, and spares once the supply train reaches the ground unit to receive the supplies.
- DISBURSE_SUPPLIES. This procedure should add the supplies to a ground unit's attributes.
- UPDATE procedures. There are numerous procedures needed to update the different attributes of a ground unit. These all begin with UPDATE and are followed by the name of the attribute which they are modifying. See Appendix F for the individual procedure names.
- WRITE_UNITS. This procedure should write out to disk the information on each ground unit.

- WRITE_SF_INTEL. This procedure should write out to disk the special forces information.
- WRITE_SUPPLY_TRAIN. This procedure should write out to disk the supply train information.

5.3.12 Hex. This package contains the procedures and functions which apply to both the air and ground hexes.

- GET_CONSTANTS. This procedure reads in from disk the user-supplied constants needed to fine-tune the hexes.
- GET_WEATHER. This procedure reads in from disk the weather information for each hex.
- CONVERT_NO. This procedure converts a hex number into three portions: LEVEL, LON, and LAT. LON and LAT were used by Ness throughout his implementation to represent x and y coordinates (or longitute and latitude). LEVEL is used to represent the levels of altitude as defined by Mann (17:57-60).
- CALC_WEATHER_VAL. This function was modified from Ness' code. It assigns a value to the four possible weather difficulties.
- CALC_TRAFFIC_VAL. This function was modified from Ness' code. It assigns a value to the six possible traffic difficulties.
- REVERSE_DIR. This function was part of Ness' implementation. It reverses by 180 degrees the direction which a unit is moving.
- UPDATE procedures. There are two procedures needed to update the different attributes of a hex. They both begin with UPDATE and are followed by the name of the attribute which they are modifying. See Appendix F for the individual procedure names.

5.3.13 OneWayLists. This package contains all the operations necessary for a list implemented as a one way linked list. The procedures and functions were taken from Feldman. (9:110) For the specific functions and procedures see Appendix F.

5.3.14 Radars. This package contains four procedures. The GET_RADARS procedure reads radar information from a disk file. The DECREASE_RADAR procedure decreases the quantity of a specific radar. The GET_CONSTANTS procedure reads in the

MAX_RADAR_QUALITY which is used to fine-tune the simulation. The last procedure, WRITE_RADARS, should write radar information out to disk.

5.3.15 Satellites. This package contains three procedures. The GET_SATELLITES procedure reads in the satellite information from a disk file. The GET_SATELLITE_QTY procedure should read in from disk the quantities of the various satellites. The DECREASE_SATELLITE procedure should decrease the quantity of a specific satellite.

5.3.16 TargetInfo. This package contains the procedures for target hardness, components, and cargo.

- GET_COMPONENTS. This procedure reads in from two disk files the data on both ground and base components.
- GET_COMPONENTS_QUANT. This procedure reads in from two disk files the quantities of the various components.
- GET_CARGO. This procedure reads in from disk the data on cargo.
- GET_CLASS. This procedure reads in from disk the component designation and the name of the corresponding database file where the component information can be found.
- DECREASE_COMPONENTS_QUANT. This procedure should reduce the quantity of a specific ground component.
- DETERMINE_SAI. This procedure should determine the surface to air index for a specific component.
- GET_HARDNESS. This procedure reads in the hardness value for each target.
- WRITE_COMPONENTS_QUANT. This procedure should write back out to disk the quantities of each component.

5.3.17 UniformPackage. This package contains a function UNIFORM which returns a uniform random number. It also contains a procedure SET_SEED which initializes the seed value required by UNIFORM.

5.3.18 Weapons. This package contains the procedures needed by the Weapons object.

- GET_SAM. This procedure reads in from a disk file the surface-to-air missile information.
- GET_SSM. This procedure reads in from a disk file the surface-to-surface missile information.
- GET_AAW. This procedure reads in from a disk file the air-to-air weapons information.
- GET_ASW. This procedure reads in from a disk file the air-to-surface weapons information.
- GET_AIR_WEAPON_QUANT. This procedure reads in the initial quantity for each air weapon.
- GET_GROUND_WEAPON_QUANT. This procedure reads in the initial quantity for each ground missile.
- GET_WEAPONS_LOAD. This procedure reads in the file which contains the various weapons load quantities.
- GET_CHEMICAL. This procedure should read in from disk the attribute information needed for the chemical type.
- GET_NUCLEAR. This procedure should read in from disk the attribute information needed for the nuclear type.
- INCREASE_WEAPONS. There are two versions of this procedure, one for the air weapons and a second for the ground missiles. Both procedures should increase the quantity for a specific weapon or missile.
- DECREASE_WEAPONS. There are two versions of this procedure, one for the air weapons and a second for the ground missiles. Both procedures should decrease the quantity for a specific weapon or missile.
- UPDATE_CEP. Four procedures are needed which update the CEP: one for the air-to-surface weapon type, a second for the surface-to-surface missile type, a third for the chemical type and a fourth for the nuclear type. All four should do nothing more than change the value of CEP using the new value passed into the procedure.
- UPDATE_WEAPON_QTY. There are two versions of this procedure. One should update the quantity for a specific air weapon while the second should update the quantity for a specific ground missile.

- UPDATE_LAUNCHER_QTY. This procedure should update the quantity of available launchers.
- WRITE_AIR_WEAPONS_QUANT. This procedure should write out to disk the quantity of each air weapon.
- WRITE_GROUND_WEAPONS_QUANT. This procedure should write out to disk the quantity of each ground weapon.

5.4 Establish the Visibility of Each Object in Relation to Other Objects

As discussed in Chapter 3, the purpose of this step is to decide which objects need to "see" and be "seen" by other objects. The "with" portion of the package specifications and body is Ada's way of showing which objects are "seen" by other objects. Appendix G shows the visibility of each object in relation to other objects. An "F" indicates the visibility of each object based on the detailed design; an "I" indicates the visibility needed based on the initial design. The visibility shown is taken from the package specifications only.

5.5 Establish the Interface of Each Object

This is the fourth of Booch's five steps and requires the writing of a module specification. This is accomplished in Ada by creating compilable package specifications. The package specifications are contained in Volume II of this thesis.

5.6 Summary

This chapter described the detailed object-oriented design based on Booch's design steps. It first described each of the objects and the rationale for making them objects. It then described the operations performed on each of the objects. It also described the visibility of each object in relation to other objects. And lastly, it described how to establish the interface of each object in Ada. The next chapter will discuss Booch's fifth and final step, "Implement Each Object".

VI. Implementation

6.1 Introduction

The final step of Booch's object-oriented design procedure is to implement each object. This chapter includes a discussion of the data structures used to implement the various objects and the programming style used for Saber as a whole. It also discusses the changes made to Ness' land module code and what was accomplished in the implementation of the air module.

6.2 Data Structures and Programming Style

This section describes the data structures used for Saber as well as the programming style.

6.2.1 Data Structures. Ness used records, arrays, and linked lists as his main data structures. Ness used records to define his ground unit and hex. He then used an array of hexes to make his hex grid. His ground units are maintained in an array of pointers. His hex grid contains an attribute which maintains a linked list of the ground units located within each hex. (20:53) An attempt was made to use the same data structures in the air module as Ness used. An attempt was also made to not modify Ness' data structures. Therefore, Ness' hex grid was left as a two-dimensional array even though the hex numbering scheme was changed. The fact that hexes now consist of three numbers did not mean Ness' code needed to be changed since the ground hexes are all level 01.

The new objects were defined as records. A list of the various records, such as aircraft, are then maintained in an array. Since an aircraft package is located within a hex just like a ground unit, a linked list was used to keep track of which air hexes contain aircraft packages. The bases and depots are located in ground hexes but because of the concerns for speed, an array construct was used to keep track of bases and depots within a specific hex.

The only variation to Ness' data structures was in the implementation of the air hex grid. Initially, it was implemented as a three-dimensional array consisting of LEVEL, LON, and LAT. This data structure meant that the array contained lots of empty records because there is only one air hex for every seven land hexes. Quite a bit of space was wasted. The data structure was then changed to a sparse matrix. This implementation

requires one to know what the maximum element size is and how many non-zero records there will be. That information was easily determined. The sparse matrix provides an array of air hexes and eliminates all the empty records thus saving space.

Once the database design was baselined, it was incorporated with the work in progress for the simulation itself. The writing of the input procedures began as did the problems. When attempting to compile the test program for the ground hex input procedure, an error was encountered which indicated that there was not enough space. The hex grid size was reduced and the test program compiled. Further testing indicated that reducing the size of the hex grid was not going to resolve the problem. The same problem occurred when testing the input procedure for the aircraft object. Changing some of the declarations from arrays to linked lists solved the problem for the aircraft, aircraft package, and bases objects. The problem for the ground hex, however, is more complicated because Ness' code is based on locating records and performing operations based on the units location within the two-dimensional hex grid. Changing the hex grid to some other data structure will require a major rework of Ness' code. Solutions to these problems are the subject of future work.

6.2.2 Programming Style. Since code is often not maintained by the person writing it, it is very important to make the code readable and understandable. This means that the programmer should not use procedure and program names that are vague nor should the programmer use any "arcane programming tricks which confuse the reader" (28:311). These concepts were kept in mind in naming the variables and procedures needed for Saber. Ness did an excellent job of naming his procedures and variables. It is apparent from his names what most of his procedures and functions do. The same type of names were given to the air portion of Saber. A procedure that is called CALC_POISSON obviously calculates a poisson distribution for something. Given a procedure name, FORM_ACPKGS, the reader can correctly guess that the procedure forms aircraft packages. The same is true with the type declarations. The reader can correctly surmise that the type declaration, SAM_TYPE, defines a surface-to-air missile while one called AIRBASE_TYPE defines an airbase.

In addition to having good procedures names it is also important for the reader to easily determine where a procedure is located. If the reader sees a procedure call, for example, INITIALIZE with some parameters, they are left to try to determine where the actual procedure is located. The only "clue" is the actual parameters. If THE_AIR_GRID

is one of the parameters, the reader might assume that the procedure is located within AirHex. In this example, the procedure is actually located within AFSim. For this reason, using the Ada "use" clause was kept to a minimum. Instead, the procedure call is explicit: ArmySim.INITIALIZE with the parameters or AFSim.INITIALIZE with the appropriate parameters.

6.3 Land Module

This section describes the reorganization of Ness' code and the changes made to it.

6.3.1 Code Reorganization. Ness' code contained many packages. He had one package which contained all of the Ada type declarations. He had other packages which performed various operations. For example, he had a package titled Combat_Ops which performed combat operations and another one called Logistic which performed logistical operations. In additon, his driver program, Main, made use of Ada's ability to separately compile procedures. His code was restructured as part of this thesis effort. The type declarations were encapsulated in packages along with the procedures and functions which performed operations on the object. For example, the hex grid declaration and the operations performed on it were placed in the GroundHex package. His "separate" procedures were encapsulated in the ArmySim package. Procedures which modified more than one object were also encapsulated in the ArmySim package. For example, Ness has a procedure called DEPOT_LOG which updates attributes of both the ground hex and the ground unit. Since, following object-oriented principles, it did not fit with either the GroundUnits or GroundHex packages, it was located in the ArmySim package. Of the packages shown in Appendix G, the ArmySim, Forces, GroundHex, GroundUnits, and Hex contain Ness' original code.

This reorganizing of Ness' code still does not make it completely object-oriented because the ArmySim package performs operations on both the ground unit and the ground hex. To continue with the DEPOT_LOG example, Ness has the following code which updates the TOTAL_AMMO attribute of a ground unit:

```
UNIT_PTR.THE_UNIT.TOTAL_AMMO := UNIT_PTR.THE_UNIT.TOTAL_AMMO + LOG(i).THE_
UNIT.TOTAL_AMMO
```

UNIT_PTR is a pointer to a particular ground unit. Therefore, any changes which occur are made to just the unit to which the pointer refers. To make Ness' code more

object-oriented, procedures were written to update attributes of the major objects. In the example given, the following code replaced Ness':

```
GroundUnits.UPDATE_TOTAL_AMMO (UNIT_PTR.THE_UNIT.TOTAL_AMMO + LOG(i).THE_
UNIT.TOTAL_AMMO)
```

By replacing the assignment statement with a procedure call, the actual change is being made within the GroundUnits package which is how it should be done according to object-oriented principles.

All of Ness' package took advantage of Ada's "use" clause. In reorganizing the code, the majority of the "use" statements were eliminated and procedure calls were made explicit.

6.3.2 Code Modifications. Ness' code was also modified to reflect the rotation of the hexes and the new numbering scheme. The definition of Ness' HEX_SIDE_TYPE, GRID_SPECS, and UNITS_TYPE were modified to include the additional features described by Mann and in Chapter 4 of this thesis. For example, Ness' type declaration permitted one obstacle per hex side and one target per unit. Those declarations were replaced with arrays which permit more than one obstacle per hex side and more than one target per unit. Ness' code was modified to read the first obstacle or target. This should be a temporary fix until the code can be reviewed and modified to correctly take advantage of the new capabilities. In addition, all references to FIREPOWER and COMBATPOWER were reversed in Ness' code. What Ness' thesis describes as FIREPOWER is now called COMBATPOWER in his code and vice versa. This is because Mann's definition of firepower was used for Saber's design.

Procedures were written to provide many enhancements to Ness' code. These procedures are located within the Algorithms package but the land module code does not currently call any of them. Examples of these procedures are: CALC_BLUE_FIREPOWER, CALC_RED_FIREPOWER, CALC_AMMO, and CALC_POL. All of these procedures were written based on Mann's Chapter 6. (17:176)

Ness had a type declaration for DIFFICULTY and subtypes of DIFFICULTY for the weather, trafficability, and obstacles. This DIFFICULTY was an enumeration type containing EXC, VG, GD, FAIR, POOR, and VP. Separate type declarations for weather and trafficability were created. It was decided that the weather should range from GD to POOR. Having three separate declarations meant that Ness' code to determine the CALC_VAL function was no longer accurate. There are now three functions that assign a numeric

value to the three types of difficulty. The CALC_VAL function located within the Ground-Hex package determines the value for the obstacle difficulty, the CALC_WEATHER_VAL and CALC_TRAFFIC_VAL are located within the Hex package and perform the same functions for weather and trafficability.

6.4 Air Module

Initially, the plan for the implementation phase of Saber included a bottom-up type implementation. The plan was to write the input procedures and then work on writing the procedures to form an aircraft package, to fly the package, and to simulate conflicts. This plan did not work as expected because of the concurrent design of the database. The effort was not wasted because in working through the functions needed to create and fly an aircraft package and to simulate conflicts, it was determined that the operations identified by the detailed design would require other operations in order to perform the functions needed.

In the meantime, another starting point was needed. This was found in the area of Mann's algorithms. Using object-oriented principles and good programming practices it was simple enough to start implementing the algorithms without knowing the database design or the final version of the objects' attributes. In fact, this technique proved beneficial because in coding the algorithms additional attributes were defined.

Chapter 5 discussed the procedures needed for the package specifications. Appendix H describes the procedures and functions which are part of the package body but not part of the package specifications. These procedures and functions are required by another procedure or function within that package. They are not required by any other packages and therefore, do not need to be in the specifications.

6.5 Summary

This chapter described the data structures used in the implementation of Saber as well as the problems encountered when testing the input procedures. Most objects are implemented via Ada's records and arrays. The air grid is implemented using a sparse matrix instead of a normal array to eliminate the wasted space caused by all the empty records.

This chapter also discussed the many changes made to the land module code as well as reorganization of the code. It discussed the reversal of the FIREPOWER and

COMBATPOWER attributes throughout Ness' code. It also describe the encapsulation of Ness' "separate" procedures into the package ArmySim.

Lastly, it described the beginning of the implementation of the air module and the processes used for the implementation. A bottom-up implementation was initially started but the method used was changed to one that centered around Mann's Chapter 6. The next chapter provides a summary, recommendations for further study, and a conclusion.

VII. Conclusion

7.1 Summary

This thesis developed an object-oriented design for the simulation portion of Saber. Included in this design were the modification of Ness' land module design and the complete design of the air module. A five step process was used for the design. The steps encompassed:

1. Identifying the objects and their attributes. This was done for the land module by reviewing Ness' thesis and Ada code. Since Ness' used good naming conventions when writing his type declarations, it was relatively easy to determine what objects he used for the implementation. Mann' thesis provided enhancements and modifications for the land module. It was also used for the selection of air module objects. This was done by considering the nouns used in Mann's thesis as possible objects. Mann's thesis and Ness' code provided the basis for the attributes of most of the objects.
2. Identifying the operations needed for each object. This was done by extracting verbs from Mann's thesis. Besides using Mann's thesis to determine operations, research group meetings on the requirements of Saber also provided additional operations. The land module's code was reviewed, as well as Ness' thesis to determine what operations were needed for the ground unit. In addition, Ness' maintenance and user's manual proved beneficial (19).
3. Establishing the visibility required between objects. This was accomplished by first looking at the attributes of the various objects and seeing what other objects where needed to define each object. It was also accomplished by looking at the operations required by an object and needed for an object.
4. Establishing the interface of each object. This was done by creating Ada package specifications and successfully compiling them.
5. Implementing each object. This is done in Ada by writing the package body. Though this step was started, due to time constraints it was not completed. An attempt was made to create procedure stubs for all the needed operations. Sound software engineering principles were used to provide easy to read and modular code.

7.2 *Ada and Simulation*

Ada has its advantages and disadvantages when used as the implementation language for a simulation. The research accomplished for this thesis indicated that some models are written using more than one language because no one language provides all the capabilities needed for a good implementation using object-oriented techniques. Ada does provides most of the capabilities needed. In fact, Ada was missing only one capability that could have been used for Saber's object-oriented implementation. The missing capability is the ability to portray inheritance. This would have been helpful, for example, in the implementation of the hex object. The air hex is a hex, the ground hex is a hex, yet they ended up being defined in separate packages and a third object, Hex, created to define common attributes. It also could have been used when defining a supply train and the surface missiles since they need the attributes of the ground unit as well as their own additional ones.

Simulations often require the use of a random number generator, statistical distributions, and a clock. The random number generator is used for such things as determining the number of missiles fired or how many aircraft enter the playing field at one time. Statistical distributions are used for many purposes. The time it takes an aircraft to land and taxi might be calculated via some distribution. In the case of Saber, statistical distributions were needed to determine the required maintenance time for an aircraft returning from a mission as well as to determine attrition. Simulations need a clock to keep track of the simulation and when events should occur. Simulation languages, like SLAM II (22), include a random number generator, various statistical distributions, and a clock but Ada does not include any of them. Therefore, a random number generator and numerous statistical distributions had to be written and a mechanism created to simulate the clock.

7.3 *Recommendations*

This thesis focused on the object-oriented design of Saber. Though a good design leads to a good implementation, time did not permit the complete implementation of Saber. The following are specific areas that require further work:

- Completion of the air module implementation. There are a number of packages which contain stubs for procedures and functions that should be completed.
- Additional research needs to be done in the area of nuclear and chemical warfare to determine the most realistic method to accomplish missions using non-conventional

warheads. What Mann's thesis describes is a good beginning but further research is needed. This research should include specific algorithms to be used in the implementation.

- Additional research should be accomplished for the use of satellites. Again, Mann's work in this area is a good start but as the military becomes more dependent on satellites, their mission becomes more important and should be realistically portrayed.
- Ness' code needs to be modified to implement the enhanced features of Saber. Currently, Saber's input provides the ability to have more than one obstacle in a ground hex. Ness' code does not permit this. The same is true with targets. Saber's design also includes rivers, roads, railroads, and pipelines; they should also be implemented in the land module. Ness' code should also be reviewed to determine if any changes need to be made to implement the concept of "neighbor_id" and to determine if the way he implements the FEBA is better than using the table provided by Horton's database.
- The ARMY_UNITS type includes Army units that have not been implemented. For example, there is a MI type that is not implemented. In addition, Saber's design added the additional capability of using surface missiles, having a supply train, and also having a predirect supply train. These additional types should be implemented.
- A student well versed in Ada and object-oriented development methods should review Ness' code and determine if a better way exists to implement the various procedures and functions. There are times when Ness repeats an entire block of code in the same procedure or in other procedures. He also updates the unit information via its link to the ground hex. This is not only difficult to follow, it is also hard for someone maintaining the code to determine that a ground unit is in fact being modified.
- Verification and validation should be accomplished before Saber is provided to the Air Force Wargaming Center. Both the land and the air modules algorithms need to be validated.
- A scenario development tool is needed to prepare the large amount of data needed to initiate a game.
- Further research needs to be done in the area of multiple-sided warfare. For example, what is the effect on Saber if there were GREEN and BLUE forces fighting RED forces.

- Further research needs to be done in the area of weapons, missiles, bombs, etc. It should be determined whether Saber's design permits correct use of the various types of weapons.
- An analysis should be done with the Saber code to determine whether it would be beneficial for Saber to run processes in parallel. This can be done in Ada using tasks instead of the current procedures.
- Saber should be enhanced to include the targeting of an air hex. This would permit the simulation of air bursts which would destroy or reduce communications capabilities.
- Ness uses an intelligence filter to vary the intelligence index for certains situations. Research should be conducted in the use of the intel filter to provide the RED player with one type of intelligence information, the BLUE player with another level of intelligence, and the controller with complete intelligence.
- A base currently can have a mission of DEPLOY. This capability should be designed and implemented. Or maybe a capability of deploying parts of a base should be considered. Certain USAF units, like Red Horse units, deploy in time of war. Aircraft and supporting personnel are deployed. It would be interesting to research whether entire bases are actually deployed.

7.4 Conclusion

In conclusion, Saber, once it is fully implemented, should provide the Air Force Wargaming Center with a viable wargame. The design provides for a flexible, easy to understand, and easy to maintain system. It also takes advantage of the software engineering principle of reusing code. It provides the user with a credible combat model for air and land warfare.

Appendix A. *Description of Ness' Thesis on the Land Battle*

This appendix summarizes Ness' thesis effort (20) which developed a new land battle module.

A.1 General

The land battle program is an aggregated model which was designed to be generic so that any combat arena could be played. The principal area of focus is on corps division and non-divisional units up to army group level. It is a stochastic model. It uses discrete events with fixed time steps.

The input source is flat files. The flat files were designed so that the input could be extracted from an Oracle data base management system. Input is also provided by the user. Output is written to flat files at the completion of a simulation run. Three portions of the land battle need to interact with an air battle module. These portions simulate reconnaissance, battlefield air interdiction, and close air support. These portions are no longer needed because of the development of the air module.

A.2 Environment

The land battle uses interlocking hexagons to represent terrain. Terrain features and obstacles, as well as weather, are simulated. Trafficability is represented both in a hex and at the boundaries of a hex. Boundaries contain bridges, mines, and manmade obstacles.

A.3 Combat Processes

The land battle simulates missions, fire and general support processes, and unit movements. The missions supported are attack, defend, withdraw, movement, and support. Logistics is modeled via the support mission; retrograde operations are simulated by the withdraw mission. Command, control, and communications are modeled only through the player input. Nuclear, chemical, and biological missions are not part of Ness' land battle.

A.4 Combat Arms Operations

The land battle simulates the activities of the following units: armor, infantry, calvary, aviation, field artillery, air defense artillery, engineering, special operations forces,

military intelligence, support, and Air Force units. It models both general support and direct support operations. Reserves can be modeled by applying them as either resupply support or as combat units.

A.5 Attrition

Attrition is assessed when opposing units occupy adjacent hexes. It is based on force ratios, engagement type, terrain characteristics, and whether the unit is attacking or defending.

Appendix B. *Description of Mann's Thesis*

This appendix summarizes Mann's thesis effort (17) on using U.S. Air Force doctrine to create a new conceptual model.

B.1 Land Module

The recently implemented land battle simulation uses hexes to facilitate the movement of ground units. Combat begins when opposing units are in adjacent hexes. The combat process is deterministic. Combat units have a firepower score which can be raised by "unit posturing, attached units, and supporting units" (17:54). Mann proposes that the land module be modified "to portray air/ground interactions" (17:54). The modification should expand the definition of the ground entities to include a counter which will be compared to the firepower score.

B.2 Environment

Saber will simulate the environment. Specifically, the new model needs the capability of providing a clock to divide a day into two hour segments. Mann proposes that the hex system used for the land module be modified for the air portion of Saber. He proposes that six layers of air hexes be superimposed over the land hexes. Since an aircraft covers a much larger area than a ground vehicle in the same amount of time, Mann proposes that an air hex should consist of seven land hexes. The seven layers of hexes will indicate the following levels of altitude:

- The terrain hex is the base hex,
- Tree top level - from 0 to 200 feet,
- Low altitude - from 200 to 2,000 feet,
- Medium altitude - from 2,000 to 10,000 feet,
- High altitude - from 10,000 to 30,000 feet,
- Very high altitude - from 30,000 to 100,000 feet, and
- Space - from 100,000 to geosynchronous orbit.

The model also needs the capability to simulate weather for each hex. Weather should be good, fair, or poor, and should vary based on the level of the hex.

B.3 Combat Processes

The combat processes which Saber should simulate include "air-to-air combat, surface-to-air missiles, suppression of enemy air defense sites, and air-to-surface attack" (17:64).

B.4 Missions

Mann's thesis describes the following missions which should be simulated by Saber:

- Strike missions:
  - Offensive Counter Air (OCA),
  - Battlefield Air Interdiction (BAI),
  - Air Interdiction (AI),
  - Close Air Support (CAS),
  - Reconnaisance (RECCE),
  - Missiles.
- Area missions:
  - Defensive Counter Air (DCA),
  - Close Air Patrol (CAP),
  - Command and Control (C2),
  - Suppresion of Enemy Air Defenses (SEAD),
  - Electronic Combat (EC),
  - Satellites,
  - Reserves.

B.5 Databases and Entities

Saber will have a number of databases. The databases will contain information on such things as aircraft, weapons, airbases, and depots for both the Blue and Red sides.

Mann's thesis provides a reference system for future modifiers of the system so they will know how he derived various values and can make any necessary modifications. He accomplishes this by modelling the individual components of entities (i.e., tanks and planes) to their engineering characteristics.

B.5.1 Ground Units. Mann's thesis has tables showing battalion equivalents for the ground units. He also describes how he calculates firepower scores for the U.S. Army's divisions, as well as separate brigades. He also provides the firepower score calculations for the Soviet Motorized Rifle Division and the Soviet Tank Division.

B.5.2 Air Defense Artillery and Missiles. "In both the US and Soviet Armies the military air defense is divided up into SHORAD and HIMAD systems" (17:100). The SHORAD is simulated by a surface-to-air index (SAI). The SAI is used along with an algorithm to determine a unit's short-range air defense against enemy aircraft. The HIMAD is a surface-to-air missile and is represented as a separate entity with its own characteristics, including a single shot probability of kill.

B.5.3 Bases. There are four main categories of bases: air bases, depots, staging bases, and missile bases. Each of these have similar characteristics: "identity, situational awareness, resources, and aircraft or missiles" (17:102). Mann describes what is contained in the database for each of the bases.

B.5.4 Aircraft. Aircraft are used in the simulation of four areas. Mann describes how to determine the combat capability of an aircraft which is used in air-to-air combat simulations. A circular error of probability is determined when calculating the air-to-ground ratings. The model uses an electronic combat value to determine the results of electronic combat. The fourth area is the determination of the resource quantities needed to fly an aircraft. The "resources include fuel, maintenance hours, spares (spare parts), ammunition, and usable runway" (17:109). If any of the items are missing, the aircraft's mission is aborted.

B.5.5 Missiles and Bombs. Mann describes the four types of missiles as follows:

> The surface-to-surface missiles are treated as aircraft. Air combat uses air-to-air missiles. The short-range surface-to-air missiles are incorporated in the SAI, whereas theater surface-to-air missiles are used to conduct combat with the aircraft packages. And finally, air-to-surface missiles are either point target destructive or an area effect weapon. (17:110-111)

Mann goes on to describe the various characteristics of each type of missile.

B.5.6 Aircraft Packages. Saber will not simulate the flying of individual aircraft but rather the usage of multiple aircraft. The multiple aircraft are formed into an aircraft package. An aircraft package should include aircraft for the primary mission as well as any electronic combat, SEAD, refueling, and escort aircraft. Aircraft package attributes are described by Mann. Attributes include the mission identification and the individual aircraft.

B.5.7 Nuclear and Chemical Weapons. Mann proposes that Saber simulate both nuclear and chemical weapons. He describes how to simulate both of these.

B.6 Overall Process

The model will read in the initial input values for a 24 hour cycle and then process the weather. As the clock changes, the computer makes any required changes to the databases. If the player's input calls for a mission to be conducted within the clock's current time period, then the simulation creates an aircraft package.

An aircraft package cannot be created unless the appropriate resources are available. The types of resources needed might include: aircraft, fuel, and weapons. After aircraft packages are formed, the simulation subtracts logistical resources and number of aircraft needed from the appropriate databases and loads ground unit data.

If there are not enough resources available for a particular mission, the mission is delayed until the next time period. The first missions to be executed are the delayed missions from the previous time period. Once all the delayed missions are accomplished, the new area and strike missions are loaded.

First, the area missions are executed in the following order: C2, EC, CAP, and DCA. "If there are conflicting missions, combat process (sic) are conducted to resolve the issues" (17:80). After the missions are resolved, the appropriate links should be in place to the air and ground hexes.

Next, the strike missions are executed in order of priority. Algorithms are used to determine the correct path across the hexes. This path becomes part of the aircraft package. "As the aircraft package enters an air hex, the computer checks to see if the package has been detected by AWACs or GCI" (17:73). The simulation then determines whether there are any Air Defense Artillery units in the ground hexes. If there are any conflicts, Saber should resolve them when they are encountered. While travelling through

the hexes, the system checks to see if the package has reached its target. Once the target is reached, the simulation is conducted.

Mann describes the following three outcomes which can result from a conflict:

> The package may be utterly destroyed, the package may have taken so many hits that it decides to abandon its mission and return home, or the package may still have sufficient combat power to continue. If the package continues and successfully arrives at the mission site, the aircraft will conduct their mission, and return to their start point using a backwards route or a return path that is recalculated. At the end point, the planes are loaded back into their bases and the appropriate supplies the aircraft have remaining are loaded back into the base or counted as consumed (17:81).

After the strike missions are accomplished, the system determines whether the area missions can continue. Any missions which can continue are kept in the area mission matrix, while others are returned to their bases. The clock is then advanced, databases updated, the ground war takes place, and the process begins again.

B.7 Algorithms

Chapter VI of Mann's thesis describes a number of algorithms which should be used in Saber.

Appendix C. *Ness' Packages and Contents*

This appendix shows the packages used by CPT Ness. It also lists the procedures, functions, and the main type declarations. For a description of the procedures and functions see Appendix F.

- AF_COMBAT. This package performs the air force operations against land units.
  - APPLY_CAS
  - APPLY_BAI
- AF_IO. This package reads in from disk all the necessary air force information.
  - GET_RECCE
  - GET_BAI
  - GET_CAS
- COMBAT_OPS. This package contains the necessary subprograms to perform required set up and combat operations of ground units. Only the first two procedures below are part of the package specifications; the remainder are only part of the package body.
  - SET_UP
  - ATTRITION
  - ASSESS_CP
  - SET_ATK
  - ASSESS_CONTACT
  - APPLY_FS
  - APPLY_CP
  - ATTRIT
  - DESTROY
    - ADDIT
  - WITHDRAW_UNIT

- CONSTANTS. This package defines all the user specified constants needed for the simulation.
  - *GET_CONSTANTS*
- DATA_IO. This package reads in from disk all the land unit information needed to run the simulation.
  - GET_GRID
  - GET_WEATHER
  - GET_UNIT
  - GET_SF_INTEL
- INTEL. This package performs land unit intelligence reduction based on loss of intelligence over time, intelligence operations of a unit's own intelligence capabilities, and military intelligence brigade operations.
  - REDUCE_INTEL
  - SF_INTEL
  - ARMY_INTEL
- LOGISTIC. This package provides all logistic and depot resupply and support operations.
  - DEPOT_LOG
  - UNIT_LOG_SUPPLY
- MAIN. The driver program. The procedures are all defined as "separate" procedures.
  - INITIALIZE
  - GET_AF
  - REPORT_WRITER
  - LOG_SPT
  - APPLY_AFS
  - MOVEMENT

- INTELLIGENCE
- AF_INTEL
- WRITE_DATA
- AIR_INTERFACE
- LOAD_WEATHER

- OBSTACLE_OPS. This package performs operations when a unit is faced by obstacles at a border. It includes the engineer brigade support and a ground unit's own inherent support.
  - OVERCOME_OBSTACLE
- PTR. This package provides the access type (pointer) operations necessary to manipulate the access types.
  - INITIAL_PTR
  - ADD
  - DELETE
- TEXT_IO
- UNIT. This package defines all the types needed by the simulation. It does not contain any functions or procedures. Listed below are the main type declarations.
  - HEX_SIDE_TYPE. The specifications for the six sides of a hexagon.
  - GRID_SPECS. The specifications applicable to an entire hexagon.
  - WEATHER_SET. An array specifying the weather difficulty by days, cycles, and weather zone.
  - COMBAT_SPT_ARRAY. An array of pointers to the units providing support.
  - ARMY_UNITS. An enumeration type declaring the various types of units.
  - UNITS_TYPE. The declaration for a ground unit.
  - SF_INTEL_RECORD. The declaration for the Special Forces intelligence units.
  - NODE. The pointer record declaration.
  - RECCE_TYPE. The record declaration for an AF reconnaissance mission.

- BAI_TYPE. The record declaration for an AF battefield air interdiction mission.
- CAS_SPT_RECORD. The record declaration showing the percentage of support being provided to a unit.
- CAS_TYPE. The record declaration for a close air support mission.

- UNIT_OPS. This package provides the basic movement and movement related operations which a ground unit must perform.
  - DETERMINE_ROUTE
  - CALC_VAL
  - UNIT_VAL
  - SELECT_ROUTE
    - CALC_DIRECTION
  - REVERSE_DIR
  - BORDER_TRANSITION
  - MOVE_IN_GRID
  - UPDATE_LOCATION
  - MANEUVER

Appendix D. *Visibility of Ness' Packages*

This appendix indicates the visibility between the packages defined by CPT Ness. This was accomplished in his code by the use of Ada "with" clauses.

OBJECTS "sees"	AF_COMBAT	AF_IO	COMBAT_OPS	CONSTANTS	DATA_IO	INTEL	LOGISTIC	MAIN	OBSTACLE_OPS	PTR	TEXT_IO	UNIT	UNIT_OPS
AF_COMBAT				X								X	
AF_IO					X						X	X	
COMBAT_OPS				X						X	X	X	X
CONSTANTS											X		
DATA_IO											X	X	
INTEL				X	X							X	
LOGISTIC				X						X		X	
MAIN			X	X	X						X	X	
OBSTACLE_OPS				X								X	
PTR												X	
TEXT_IO													
UNIT													
UNIT_OPS				X					X	X	X	X	

Appendix E. *Objects and their Attributes*

This appendix shows the various objects followed by their attributes. Attributes shown in italics are a result of the detailed design. Most of the attributes are described in Horton's data dictionary (11). The attributes not described in his data dictionary are described after their usage. Attributes which have a different name here than in Horton's data dictionary have the corresponding data dictionary name following them.

- AFSim
- AIRCRAFT
  - DESIGNATION
  - FORCE
  - *NIGHT_CAP* - Horton calls this attribute night_capability.
  - WEATHER_CAP - Horton calls this attribute wx_capability.
  - COMBAT - Horton calls this attribute a2a_rating; Mann calls it COMBAT.
  - *SIZE* - Horton calls this attribute ac_size.
  - AVG_SORTIES_PER_WEEK - Horton calls this attribute sorties_week.
  - SEARCH
  - EC
  - MAX_SPEED
  - COMBAT_RADIUS - Horton calls this attribute radius.
  - LOITER_TIME
  - CARGO
  - RECON_ABILITY
  - REFUEL - Horton calls this attribute refuelable.
  - *MAINT_DIST* - Horton calls this attribute maintain_dist.
  - *MAINT_MEAN* - Horton calls this attribute maintain_mean.
  - *MAINT_STAND_DEV* - Horton calls this attribute maintain_standev.
  - AMT_SPARES - Horton calls this attribute spare_parts.

- POL - Horton calls this attribute pol_usage_rate.
- RAMP - Horton calls this attribute ramp_space.
- MIN_RUNWAY_NEEDED - Horton calls this attribute min_runway.
- PCL - This is the preferred conventional load. It contains the designations and quantities of weapons needed for a specific aircraft for all missions which the plane can fly. A PCL record contains the following attributes:
  * MISSION - Horton calls this attribute mission_type.
  * TARGET_HARDNESS - Horton calls this attribute hardness.
  * PRIM_LOAD - This attribute is the integer part of Horton's load_id which applies to the primary weapons load.
  * SECOND_LOAD - This attribute is the integer part of Horton's load_id which applies to the secondary weapons load.
  * TERTIARY_LOAD - This attribute is the integer part of Horton's load_id which applies to the tertiary weapons load.
- *PBL* - This is the preferred biological/chemical load. It contains the designations and quantities of weapons needed for a specific aircraft carrying a chemical warhead for all missions which the plane can fly. Each record has the same attributes as a PCL record.
- *PNL* - This is the preferred nuclear load. It contains the designations and quantities of weapons needed for a specific aircraft carrying a nuclear warhead for all missions which the plane can fly. Each record has the same attributes as a PCL record.
- AIR_GROUND_RATING - Horton calls this attribute a2g_rating.
- MAX_HEX_LEVEL - Horton calls this attribute max_hex.

- AircraftPackage
  - MISSION_NO - Horton calls this attribute mission_id.
  - FORCE
  - PRIMARY_MISSION - Horton calls this attribute mission_type.
  - TARGET - This is the number of a target. It is the ASSET_ID.
  - RQST_PRD_ON_TARGET

- RQST_DAY_ON_TARGET
- ACTUAL_START_PRD
- ACTUAL_START_DAY
- LOITER_TIME
- *RQST_RETURN_PRD*
- *RQST_RETURN_DAY*
- ACTUAL_RETURN_PRD
- ACTUAL_RETURN_DAY
- PRIORITY
- ACTIVATED
- RENDEZVOUS_HEX
- DISTANCE - This is the total distance the aircraft package can fly. It is based on the aircraft in the package which flies the shortest distance.
- ALTITUDE - This is the highest hex level which the aircraft package can fly and is based on the capabilities of the aircraft which are part of the package.
- SPEED - This is the speed of the slowest aircraft in the package and indicates the overall speed of the aircraft package.
- INEFFECTIVE_REASON
- ORBIT_LOCATION
- ATTRIT_PER_AIR_HEX - This is the amount of attrition for each air hex the aircraft package flies through.
- DETECTED - This is a boolean value which reflects true when the aircraft package is detected by the enemy's early warning systems.
- POSITIVE_ID - This boolean value indicates that not only was the aircraft package detected but it was also positively identified by the enemy.
- WAS_CANCELLED - This boolean value indicates that the aircraft package was cancelled.
- *WARHEAD* - This attribute indicates whether the warhead is chemical, nuclear, or conventional.

- PRIM_AC_SCHED - This attribute provides a list of the primary scheduled aircraft and their quantities.
- PRIM_MSN_AC_START_NO - This attribute provides a list of the primary aircraft and quantities which actually started on a mission.
- PRIM_MSN_AC_PRES_NO - This attribute provides a list of the present primary aircraft and quantities. It is used to indicate how many aircraft return from a mission.
- ESCORT_OR_CAP_AC_SCHED - This attribute provides a list of the escort and close air patrol (CAP) aircraft that are being scheduled by the user to fly a mission. This information comes from Horton's AIRCRAFT_PACKAGE relation.
- ESCORT_OR_CAP_AC_START_NO - This attribute provides a list of the escort and CAP aircraft that actually started on a mission.
- ESCORT_OR_CAP_AC_PRES_NO - This attribute provides a list of the escort and CAP aircraft that are currently part of the aircraft package. This number is reduced when an escort or CAP aircraft is destroyed.
- SEAD_AC_SCHED - This attribute provides a list of the SEAD aircraft scheduled by the user to be flown as support aircraft as part of an aircraft package. This information comes from Horton's AIRCRAFT_PACKAGE relation.
- SEAD_AC_START_NO - This attribute provides the actual starting number of SEAD aircraft flying a support mission.
- SEAD_AC_PRES_NO - This attribute provides the present number of SEAD aircraft on a mission.
- ECM_AC_SCHED - This attribute provides a list of the electronic countermeasure (ECM) aircraft which the user has requested for a support role within an aircraft package. This information comes from Horton's AIRCRAFT_PACKAGE relation.
- ECM_AC_START_NO - This attribute provides a list of the actual starting number of ECM aircraft for a particular aircraft package.
- ECM_AC_PRES_NO - This attribute provides a list of the present number of ECM aircraft returning from, or during, a mission.

- REFUEL_AC_SCHED - This attribute provides a list of the refueling aircraft which the user has requested for a support role within an aircraft package. This information comes from Horton's AIRCRAFT_PACKAGE relation.
- REFUEL_AC_START_NO - This attribute provides a list of the actual starting number of refueling aircraft for a particular aircraft package.
- REFUEL_AC_PRES_NO - This attribute provides a list of the present number of refueling aircraft returning from, or during, a mission.

- *AirHex*
  - WEATHER - Horton calls this attribute actual_wx.
  - WEATHER_ZONE - Horton calls this attribute wz.
  - ATTRITION
  - EC
  - *TRAFFICABILITY*
  - *PERSISTENCE* - Horton calls this attribute persistence_time.
  - *NEXT_ACPKG* - This attribute provides a linked list of aircraft packages located in a particular air hex.
- ALGORITHMS
- ArmySim
- BASES (Includes Air Bases and Depots)
  - *ASSET_ID* - Horton calls this an airbase_id.
  - FORCE
  - *HQ*
  - MOVE_ALLOWED - This is a boolean value and indicates whether the base can deploy or not.
  - MISSION - Horton calls this attribute base_mission.
  - PRES_LOC - Horton calls this attribute location.
  - FUT_LOC - Horton calls this attribute future_location.
  - WIDTH

- LENGTH
- *WEATHER_MIN* - Horton calls this attribute weather_minimum.
- IS_BASE_OVERRUN - This is a boolean value that indicates whether the base or depot is overrun.
- IS_BASE_WITHIN_ENEMY_ART - This is a boolean value which indicates whether the base or depot is within the range of enemy artillery.
- IS_BASE_UNDER_NUC_CHEM_ATK - This is a boolean value that indicates whether the base is under chemical or nuclear attack.
- ACTIVE_ENEMY_MINES - Horton calls this attribute enemy_mines.
- MOPP_POSTURE
- IS_BASE_UNDER_AIR_ATK - This is a boolean value that indicates whether the base or depot is under air attack.
- ALT_BASES - This attribute provides a list of alternate bases where aircraft can land if the runway on their home base is no longer of sufficient size for a landing. This list is taken from Horton's ALTERNATE_BASES relation.
- POL_SOFT_STORE
- POL_HARD_STORE
- *MAX_POL_SOFT*
- *MAX_POL_HARD*
- MAINT_PERS_ON_HAND - Horton calls this attribute maint_personnel.
- MAINT_HRS_ACCUM
- MAINT_EQUIP_ON_HAND - Horton calls this attribute maint_equip.
- SPARE_PARTS
- RUNWAYS - This attribute is a list of available runways on a specific base. It comes from Horton's RUNWAYS relation and consists of RUNWAY, CONDITION (difficulty according to Horton), CURRENT_LENGTH, and MAX_LENGTH.
- MAX_RAMP_SPACE
- *RAMP_AVAIL*

- SHELTERS
- EOD_CREWS
- RRR_CREWS
- WEAPONS_AVAIL - This attribute provides a list of weapons and quantities based on the weapons' designation.
- AC_TYPES_NOS - This attribute provides a list of aircraft and their quantities.
- *AC_MAINT* - This attribute provides a list of aircraft currently undergoing maintenance. The values are calculated by the simulation but eventually get read into Horton's MAINTENANCE relation. Each AC_MAINT_TYPE record contains the following attributes:
  * INDEX - This is an integer which refers to the position in the aircraft array for a particular aircraft.
  * CURRENT_QUANT - Horton calls this attribute quantity.
  * MAINT_TIME
  * START_TIME
- TOTAL_AC - This attribute is the total number of aircraft on the base.
- STATUS
- NO_TIMES_ATCK - This attribute indicates the total number of times a base has been attacked.
- INTEL_INDEX

- *Clock*
  - *CYCLE_TYPE*
    * *CYCLE*
    * *PERIOD*

- *Forces*
  - FORCES_TYPE is (BLUE, RED, NEUTRAL)
  - *COUNTRY_TYPE*
    * *COUNTRY*

    * *FORCE*

- *GroundHex*
    - GRID_SPECS
        * WEATHER - Horton calls this attribute actual_wx.
        * WEATHER_ZONE - Horton calls this attribute wz.
        * FORCE
        * SIDE_DEF - This attribute is an array containing information on the six sides of a hex.
        * MISSION - Horton calls this attribute an army_mission_type.
        * IN_CONTACT - This is a boolean value indicating whether a ground unit is in contact with the enemy.
        * IN_ATTRITION - This is a boolean value indicating whether a ground unit is in attrition with the enemy in adjacent hexes.
        * FP_OUT - Horton calls this attribute cpo.
        * FP_IN - Horton calls this attribute cpi.
        * ATTRITION - This is a rate used by the land module.
        * SAI - This is the aggregated surface-to-air index for the units within a hex.
        * INTEL_INDEX
        * NEXT_UNIT - This attribute is a pointer to the ground units located within the hex.
        * *PERSISTENCE* - Horton calls this attribute persistence_time.
        * EC
        * *CENTER_HEX*
        * *BASE_LIST* - This attribute is a list of airbases and depots currently located in the hex.
    - HEX_SIDE_TYPE
        * NEIGHBOR_NO - Horton calls this attribute neighbor_id.
        * OBSTACLES - This attribute is a list of obstacles and is taken from Horton's HEXSIDE_ASSETS table. Each record consists of the following: OBSTACLE_NO, OBSTACLE, and OBS_DIFF. Horton calls these attributes obstacle_id, obstacle, and difficulty.

    * TRAFFIC - This attribute is taken from Horton's TRAVEL table. It equates to Horton's pie_trafficability.
    * FEBA - This boolean value is based on Horton's FEBA table.
    * BORDER - This boolean value is based on Horton's BORDERS table.
    * COAST - This boolean value is based on Horton's COASTS table.
    * RIVER_SIZE
    * ROADS - This attribute is a list of roads going into the hex. It is taken from Horton's ROADS table. It has the following attributes:
        · ROAD_NO - Horton calls this attribute road_id.
        · SIZE - Horton calls this attribute road_size.
        · FLOW
    * RAILROADS
        · RAILROAD_NO - Horton calls this attribute railroad_id.
        · FLOW
    * PIPELINES
        · PIPELINE_NO - Horton calls this attribute pipeline_id.
        · FLOW

- GroundUnits
    - *ASSET_ID* - Horton calls this attribute unit_id.
    - CORPS_ID
    - *PARENT_UNIT*
    - UNIT_SIZE
    - TYPE_OF_UNIT - Horton calls this attribute unit_type.
    - FORCE
    - PRESENT_LOCATION - Horton calls this attribute location.
    - MSN_EFF_DAY
    - REGION
    - HEX_DIR - This attribute indicates the direction of movement for a unit.
    - MOVE_ALLOWED - This attribute is an array of authorized side movements.

- IN_ATTRITION - This boolean value indicates whether a unit is in attrition with other units.
- FIREPOWER
- COMBATPOWER - Horton calls this attribute combat_power.
- ATTRITION
- TOTAL_POL
- POL_RESUPPLY_PERCENT - Horton calls this attribute pol_resupply_pct.
- *POL_USAGE_RATE*
- TOTAL_AMMO
- AMMO_RESUPPLY_PERCENT - Horton calls this attribute ammo_resupply_pct.
- *AMMO_USAGE_RATE*
- TOTAL_HARDWARE
- HARDWARE_RESUPPLY_PERCENT - Horton calls this attribute hw_resupply_pct.
- *HARDWARE_USAGE_RATE* - Horton calls this attribute hw_usage_rate.
- DEPOT_SPT - This attribute is a pointer to the depot supporting a particular unit.
- IN_CONTACT - This boolean value indicates that the unit is in contact with other units.
- INTEL_INDEX
- INTEL_FILTER
- WAS_INTELED - This boolean value indicates whether the unit has been intelled or not.
- BREAKPT - Horton calls this attribute breakpoint.
- GRID_TIME
- IS_CS - This boolean value indicates whether a unit is providing combat support or not.
- IS_INTERDICTION - This boolean value indicates whether a unit is an interdiction unit or not.

- SPTED_UNITS - This attribute is a list of support units and is taken from Horton's UNIT_SUPPORTS relation. Each individual record has two attributes as defined by Ness: UNITS_TO_SPT (Horton calls it UNIT_SUPPORTED_ID) and SPT_PERCENT (Horton calls it percent).
- *DAY_LAST_INTELLED*
- *PRD_LAST_INTELLED*
- *LOC_LAST_INTELLED*
- MISSION - This attribute is a list of missions taken from Horton's MOVE table. This record is defined by the simulation as ORDER_TYPE and consists of:
  * TARGET_NO - Horton calls this attribute target_id.
  * ORDER_NO - Horton calls this attribute order_id.
  * DAY
  * PERIOD
  * MISSION - Horton calls this attribute army_mission_type.
- *OVERRIDE_MISSION* - This attribute is is a list of missions and is taken from Horton's MOVE_LNLT table. The OVERRIDE_MISSION attribute list contains the same attributes as the MISSION attribute.
- UNDER_CHEM_NUC_ATK - This boolean value indicates whether the unit is under chemical or nuclear attack.
- MOPP_POSTURE
- TROOP_QUALITY
- ASSETS - This attribute is a list of ground components and quantities. It is taken from Horton's UNIT_COMPONENTS table.
- *SSM* - This attribute is a list of surface-to-surface missiles assigned to the unit along with the quantity of each missile. It is extracted from Horton's UNIT_S2S table. It also includes the quantity of the launchers used for each missile.
- *SAM* - This attribute is a list of surface-to-surface missiles assigned to the unit along with the quantity of each missile. It is extracted from Horton's UNIT_S2A table. It also includes the quantity of the launchers used for each missile.
- *RADAR* - This attribute is a list of radars assigned to a unit. See the Radar object for the radar's attributes.

- *GROUNDSPEED*
- PERCENT_SAI_USED - This attribute is an integer indicating the percentage of surface-to-air index used by the unit.
- FUEL_TRUCKS
- AMMO_TRUCKS
- WATER
- WATER_PERCENT
- WATER_TRUCKS
- ENGINEERS
- ENG_VEHICLES
- *STATUS*

- SupplyTrain
  - *GroundUnits attributes*
  - *MISSION_INFO* - This attribute is a list of supply missions and is taken from Horton's SUPPLY_MOVEMENT relation. It includes the following attributes:
    - TARGET_NO - Horton calls this attribute target_id.
    - ORDER_NO - Horton calls this attribute order_id.
    - SUPPLIES_DELIVER - Horton calls this attribute designation.
    - DELIVERY_QUANTITY - Horton calls this attribute deliver_qty.
  - TOT_CAP - Horton calls this attribute total_capacity.
  - IN_USE
  - *SUPPLY* - Horton calls this attribute supply_type.
  - TRANS_MODE
  - TOTAL_POL
  - TOTAL_AMMO
  - *TOTAL_HARDWARE*
  - TOTAL_SPARES - Horton calls this attribute spare_parts.

- Hey

- *WEATHER_FORECAST*
    - *GOOD_PERCENT* - Horton calls this attribute forecast_good.
    - *FAIR_PERCENT* - Horton calls this attribute forecast_fair.
    - *WEATHER* - Horton calls this attribute actual_wx.

- *OneWayList*
    - This is a generic package which uses a user defined InfoType.

- *Radars*
    - *TYPE_RADAR* - Horton calls this attribute radar_type.
    - *QUALITY*
    - *QUANTITY*

- Satellites
    - ASSET_ID - Horton calls this attribute satellite_id.
    - *DESIGNATION*
    - FORCE
    - LOCATION
    - SAT_TYPE
    - STATUS
    - *SPEED*
    - *DIRECTION*
    - *ORBIT*
    - *SAT_DELAY* - Horton calls this attribute delay.

- Targets
    - *CARGO_CAPACITY_TYPE*
        - *CARGO* - Horton calls this attribute vehicle.
        - *CAPACITY*
    - *DESIGNATION_TYPE*

    * *DESIGNATION*
    * *QUANTITY* - Horton calls this attribute weapon_count.
  - *GROUND_COMPONENTS_TYPE*
    * *DESIGNATION*
    * *AMMO_USAGE_RATE*
    * *POL_USAGE_RATE*
    * *HARDWARE_USAGE_RATE* - Horton calls this attribute hw_usage_rate.
    * *TARGET_WEIGHT* - Horton calls this attribute target_wgt.
    * *FIREPOWER_WEIGHT* - Horton calls this attribute firepower.
    * *LENGTH*
    * *WIDTH*
  - *BASE_COMPONENTS_TYPE*
    * *DESIGNATION*
    * *TARGET_WEIGHT* - Horton calls this attribute target_wgt.
    * *LENGTH*
    * *WIDTH*
  - *CLASS_TYPE*
    * *DESIGNATION*
    * *RELATION*
  - *HARDNESS_TYPE*
    * *TARGET* - Horton calls this attribute target_type.
    * *PK_VALUE* - Horton calls this attribute hardness.

- UniformPackage
  - This is a generic package which permits the user to define the range of float numbers desired.

- WEAPONS
  - AIR_TO_AIR_WEAPONS
    * DESIGNATION

    * *FORCE*
    * MISS_RANGE - Horton calls this attribute range.
    * SSPK
- AIR_TO_SURFACE_WEAPONS
    * DESIGNATION
    * *FORCE*
    * LETHALITY_RADIUS - Horton calls this attribute lethal_area.
    * CEP
    * PK_HARD_POINT_TYPE - Horton calls this attribute pk_hard.
    * PK_MED_POINT_TYPE - Horton calls this attribute pk_med.
    * PK_SOFT_POINT_TYPE - Horton calls this attribute pk_soft.
- SURFACE_TO_SURFACE_MISSILE
    * *GroundUnits attributes*
    * *DESIGNATION*
    * *FORCE*
    * *WARHEAD* - Horton calls this attribute class.
    * *LETHALITY_RADIUS* - Horton calls this attribute lethal_area.
    * *CEP*
    * *PK_HARD_POINT_TYPE* - Horton calls this attribute pk_hard.
    * *PK_MED_POINT_TYPE* - Horton calls this attribute pk_med.
    * *PK_SOFT_POINT_TYPE* - Horton calls this attribute pk_soft.
    * *MIN_RANGE*
    * *MAX_RANGE*
    * *LAUNCHER_ROUNDS* - Horton calls this attribute rnds_per_launcher.
    * *RELOAD_TIME*
- SURFACE_TO_AIR_MISSILE
    * GroundUnits attributes
    * *DESIGNATION*
    * *FORCE*
    * *WARHEAD* - Horton calls this attribute class.

    * *SLOW_HIGH*
    * *SLOW_LOW*
    * *FAST_HIGH*
    * *FAST_LOW*
    * SSPK
    * MISS_RADAR_RANGE - This is a list of both the radar's range and the missile's range for each missile by hex levels. It consists of Horton's radar2, radar3, radar4, radar5, radar6, and radar7. It also consists of Horton's range2, range3, range4, range5, range6, and range7.
    * *LAUNCHER_ROUNDS* - Horton calls this attribute rnds_per_launcher.
    * RELOAD_TIME
    * WEATHER_MIN
- CHEMICAL
    * DESIGNATION
    * *FORCE*
    * PERSISTENCE - Horton calls this attribute persistence_time.
    * LETHALITY
    * CEP
- NUCLEAR
    * DESIGNATION
    * YIELD
    * *FORCE*
    * CEP
    * *PERSISTENCE* - Horton calls this attribute persistence_time.

Appendix F. *Objects and their Operations*

This appendix shows the objects and the operations required for the object. Items in italics were not part of the initial design.

- *AFSim*
  - *INITIALIZE*
  - *PERFORM_MISSIONS*
  - *RUN_DELAYED_MISSIONS*
  - *WRITE_DATA*
  - *REPORT_WRITER*
- Aircraft
  - GET_AC
  - INCREASE_AC
  - DECREASE_AC
  - *ADD_TO_MAINT_LIST*
  - *DELETE_FROM_MAINT_LIST*
  - *DETERMINE_PCL*
  - *DETERMINE_PBL*
  - *DETERMINE_PNL*
  - *GET_AC_QUANT*
  - *GET_REFUEL_CAP*
  - *GET_MAINT*
  - *WRITE_MAINT*
  - *WRITE_AC_QUANT*
- AircraftPackage
  - *GET_ACPKG_INFO*

- FORM_ACPKGS
- BREAKUP_ACPKG
- *DELAY_MISSION*
- *GET_CONSTANT*
- DECREASE_ASSETS
- *DELETE_ACPKG_MISSION_LIST*
- *DELETE_ACPKG_DELAY_LIST*
- *UPDATE_DETECTED*
- *UPDATE_POSITIVE_ID*
- *UPDATE_ACTIVATED*
- *UPDATE_ACTUAL_RETURN*
- *UPDATE_INTEL_INDEX*
- *UPDATE_INEFFECTIVE_REASON*
- *UPDATE_PACKAGE_EC*
- *UPDATE_PRIM_MSN_AC_START_NO*
- *UPDATE_PRIM_MSN_AC_PRES_NO*
- *UPDATE_ESC_OR_CAP_START_NO*
- *UPDATE_ESC_OR_CAP_PRES_NO*
- *UPDATE_SEAD_START_NO*
- *UPDATE_SEAD_PRES_NO*
- *UPDATE_ECM_START_NO*
- *UPDATE_ECM_PRES_NO*
- *UPDATE_REFUEL_START_NO*
- *UPDATE_REFUEL_PRES_NO*
- *WRITE_ACPKG_INFO*
- *WRITE_DELAY_MISSIONS*

- *AirHex*

- *GET_AIR_GRID*

  LOAD_WEATHER

- *ADD_ACPKG*
- *DELETE_ACPKG*
- *UPDATE_LOCATION*
- *WRITE_AIR_GRID*

- Algorithms
  - *DETERMINE_DIRECTION*
  - CALC_LOCAL_DETECTION
  - ONE_MISS_ONE_AC
  - MULT_MISS_ONE_AC
  - MULT_MISS_MULT_AC
  - CALC_NO_MISS_FIRED
  - *CALC_BERNOULLI*
  - *CALC_BINOMIAL*
  - *CALC_POISSON*
  - *CALC_NORMAL*
  - CALC_PROB_HIT_CIRC_TARGET
  - CALC_PROB_HIT_RECT_TARGET
  - *CALC_BLUE_FIREPOWER*
  - *CALC_RED_FIREPOWER*
  - *CALC_POL*
  - *CALC_AMMO*
  - *CALC_PROB_LAUNCH*
  - *CALC_SSPK*
  - CALC_PROB_KILL
  - DETERMINE_PROB_FOR_TARGETS

  - CALC_ADA_PK
  - DETERMINE_LONG_RANGE_DETECTION

- ArmySim
  - INITIALIZE
  - REPORT_WRITER
  - LOG_SPT
  - MOVEMENT
  - INTELLIGENCE
  - WRITE_DATA
  - SET_UP
  - ATTRITION
  - PERFORM_ST_OPS

- Bases
  - GET_BASE
  - GET_DEPOT
  - INCREASE_RESOURCES
  - DECREASE_RESOURCES
  - *INCREASE_SUPPLIES*
  - *DECREASE_MAINT_NOS*
  - *UPDATE_IS_OVERRUN*
  - *UPDATE_IS_WITHIN_ENEMY_ART*
  - *UPDATE_IS_UNDER_NUC_CHEM_ATK*
  - *UPDATE_UNDER_AIR_ATK*
  - *UPDATE_RUNWAY*
  - *UPDATE_RRR_CREWS*
  - *UPDATE_EOD_CREWS*
  - *UPDATE_SHELTERS*

- *UPDATE_POL_HARD*
- *UPDATE_POL_SOFT*
- *UPDATE_MOVE_ALLOWED*
- *UPDATE_BASE_DIMENSIONS*
- *UPDATE_INTEL_INDEX*
- WRITE_BASES
- WRITE_DEPOTS

- *Clock*
  - *GET_CONSTANTS*
  - *GET_CYCLES*

- Forces
  - *GET_COUNTRIES*

- GroundHex
  - GET_GRID
  - *GET_CONSTANTS*
  - *CALC_VAL*
  - *INITIAL_PTR*
  - *ADD*
  - *DELETE*
  - *ADD_BASE*
  - *DELETE_BASE*
  - *LOAD_WEATHER*
  - ASSESS_CONTACT
  - APPLY_FS
  - APPLY_FP
  - ATTRIT

- SF_INTEL
- UPDATE_OBSTACLE
- UPDATE_OBS_DIFF
- UPDATE_OBSTACLE_REC
- UPDATE_INTEL_INDEX
- *UPDATE_MISSION*
- *UPDATE_IN_CONTACT*
- *UPDATE_IN_ATTRITION*
- *UPDATE_ATTRITION*
- *UPDATE_FP_IN_OUT*
- *UPDATE_FEBA*
- UPDATE_EC
- UPDATE_TRAFFIC
- *UPDATE_PIPELINE*
- *UPDATE_ROAD*
- *UPDATE_RAILROAD*
- *UPDATE_RIVER*
- WRITE_GRID

- GroundUnits
  - GET_CONSTANTS
  - GET_UNIT
  - GET_SF_INTEL
  - *GET_SUPPLY_TRAIN*
  - ADDIT
  - UNIT_LOG_SUPPLY
  - REDUCE_INTEL
  - UNIT_VAL

- MOVE_IN_GRID
- *DELETE_SUPPLY_TRAIN*
- *INCREASE_ST_ASSETS*
- *DISBURSE_SUPPLIES*
- *UPDATE_ATTRITION*
- UPDATE_TOTAL_AMMO
- *UPDATE_COMBATPOWER*
- *UPDATE_INTEL_INDEX*
- *UPDATE_DEPOT_SPT*
- *UPDATE_UNITS_TO_SPT*
- *UPDATE_IS_CS*
- UPDATE_TOTAL_POL
- UPDATE_TOTAL_HARDWARE
- *UPDATE_GRID_TIME*
- *UPDATE_WAS_INTELED*
- *UPDATE_HEX_DIR*
- *UPDATE_REGION*
- *UPDATE_FIREPOWER*
- *UPDATE_IN_CONTACT*
- *UPDATE_IN_ATTRITION*
- *UPDATE_DESTROYED_LIST*
- *UPDATE_MISSION*
- *UPDATE_MOVEMENT_TIME*
- WRITE_UNITS
- WRITE_SF_INTEL
- WRITE_SUPPLY_TRAIN

- Hex

- GET_CONSTANTS
- GET_WEATHER
- *CONVERT_NO*
- CALC_WEATHER_VAL
- CALC_TRAFFIC_VAL
- REVERSE_DIR
- *UPDATE_MOVE_ALLOWED*
- *UPDATE_LOCATION*

- *OnyWayLists*
  - *MakeNode*
  - *InfoPart*
  - *Front*
  - *Empty*
  - *Successor*
  - *Predecessor*
  - *AddToFront*
  - *AddToRear*
  - *InsertBefore*
  - *InsertAfter*
  - *Delete*

- Radars
  - *GET_RADARS*
  - *DECREASE_RADAR*
  - *GET_CONSTANTS*
  - *WRITE_RADARS*

- Satellites

- GET_SATELLITES

- TargetInfo
  - *GET_COMPONENTS*
  - *GET_COMPONENTS_QUANT*
  - *GET_CARGO*
  - *GET_CLASS*
  - *DECREASE_COMPONENTS_QUANT*
  - *DETERMINE_SAI*
  - *GET_HARDNESS*
  - *WRITE_COMPONENTS_QUANT*

- UniformPackage
  - UNIFORM
  - *SET_SEED*

- Weapons
  - GET_SAM
  - GET_SSM
  - GET_AAW
  - GET_ASW
  - *GET_AIR_WEAPONS_QUANT*
  - *GET_GROUND_WEAPONS_QUANT*
  - *GET_WEAPONS_LOAD*
  - GET_CHEMICAL
  - GET_NUCLEAR
  - INCREASE_WEAPONS
  - DECREASE_WEAPONS
  - INCREASE_LAUNCHERS

- DECREASE_LAUNCHERS
- *UPDATE_CEP*
- *UPDATE_WEAPON_QTY*
- *UPDATE_LAUNCHER_QTY*
- *WRITE_AIR_WEAPONS_QTY*
- *WRITE_GROUND_WEAPONS_QTY*

Appendix G. *Visibility of Objects*

This appendix indicates which objects "see" other objects and which objects are "seen" by other objects. An "I" indicates the visibility noted by the initial design; an "F" indicates the visibility which resulted from the detailed design and preparation of the Ada package specifications. An "X" indicates the visibility required by the Ada package body. Two packages are not shown in the table. The Algorithms package uses Verdix Ada's Math package. The TEXT_IO package is used by every object.

OBJECTS "sees"	AFSim	Aircraft	AircraftPackage	AirHex	Algorithms	ArmySim	Bases	Clock	Forces	GroundHex	GroundUnits	Hex	OneWayLists	Radars	Satellites	TargetInfo	UniformPackage	Weapons
AFSim		X	F	F	X		F	X	X	F	X	X			F	X		X
Aircraft									F			F	F			F		F
AircraftPackage		IF		I			F		IF			IF	F					IF
AirHex			F									F			F			
Algorithms																	X	
ArmySim					X			X	X	F	F	X				X	X	X
Bases		IF							IF		I	IF	F					IF
Clock																		
Forces																		
GroundHex							F	X	F		F	F	F					
GroundUnits								X	F	I		IF		F		F		F
Hex								F									X	
OneWayLists																		
Radars																		
Satellites				I			I		IF		I	IF						
TargetInfo																		
UniformPackage																		
Weapons									F			F						

Appendix H. *Additional Operations Needed*

This appendix describes the procedures and functions which are part of the package body but not part of the package specifications. These procedures and functions are required by another procedure or function within that package. They are not required by any other package and, therefore, do not need to be in the specifications. For the sake of consistency, this appendix includes the additional procedures which were part of Ness' original code.

- AFSim.
  - JETTISON_WEAPONS. This procedure should release from the aircraft package any weapons not used on a mission. It is needed by the CONTINUE_MISSION and RUN_STRIKE_MISSIONS procedures.
  - ACCOMPLISH_STRIKE_MISSION. This procedure is called by RUN_STRIKE_MISSIONS if an aircraft package makes it to its target. It in turn calls the correct targeting procedure.
  - ACCOMPLISH_SUPPORT_MISSION. This procedure is called by RUN_SUPPORT_MISSIONS if an aircraft package makes it to the location of its target. It in turn calls the correct targeting procedure.
  - APPLY_CHEMICAL. This procedure should simulate the use of a chemical weapon. It should implement what Mann describes on pages 166-174 of his thesis (17).
  - APPLY_NUCLEAR. This procedure should simulate the use of a nuclear weapon. It should implement what Mann describes on pages 160-165 of his thesis (17).
  - PERFORM_SATELLITE_OPS. This procedure should perform satellite operations as described by Mann on pages 170-174 of his thesis (17).
  - REFUEL_AIRCRAFT. This procedure should refuel aircraft if refueling aircraft are part of the aircraft package.

- DETERMINE_TARGETS. This procedure is used by a reconnaissance mission. It should keep track of what targets the reconnaissance aircraft "sees" as it flies through the hexes (17:175). It is called by RUN_SUPPORT_MISSIONS, RUN_STRIKE_MISSIONS, and RUN_AREA_MISSIONS.
- TARGET_BASE. This procedure is called by ACCOMPLISH_STRIKE_MISSION and is used when the target is a base or depot. It should determine what is located on the base that can be targeted. It should use Mann's targeting algorithm (17:156-160).
- TARGET_GROUND_UNIT. This procedure is called by ACCOMPLISH_STRIKE_MISSION and is used when the target is a ground unit. It should determine what is located with the ground unit that can be targeted. It should use Mann's targeting algorithm (17:156-160).
- TARGET_GROUND_HEX. This procedure is called by ACCOMPLISH_STRIKE_MISSION and is used when the target is a ground hex. It should determine what is located within the ground hex that can be targeted. It should use Mann's targeting algorithm (17:156-160).
- TARGET_OBSTACLE. This procedure is called by ACCOMPLISH_STRIKE_MISSION and is used when the target is an obstacle. It should determine the correct targeting algorithm to use and call the correct procedure. For example, if the obstacle is a bridge, then the rectangular target algorithm would be used to determine how much destruction occurred.
- TARGET_SUPPLY_TRAIN. This procedure is called by ACCOMPLISH_STRIKE_MISSION and is used when the target is a supply train. It should determine what the supply train encompasses that can be targeted. It should use Mann's targeting algorithm (17:156-160).
- RUN_ADA. This procedure runs the air defense artillery by using the surface-to-air index of a ground unit (17:135- 141).
- DETERMINE_DETECT_SAM. This procedure should determine whether an aircraft package is detected by a surface-to-air missile. If the aircraft is de-

tected, then the amount of damage is determined. It is called by RESOLVE_CONFLICTS and, in turn, calls functions located within the Algorithms package.

- DETERMINE_LONG_RANGE_DETECT. This procedure should determine if an aircraft package is detected by any airborne early warning aircraft (AWACs) or ground control intercept (GCI) aircraft. It is called by RESOLVE_CONFLICTS and, in turn, calls functions located within the Algorithms package.
- RUN_AIR_TO_AIR. This procedure should perform air-to-air combat if opposing aircraft packages are located in the same air hex (17:142-146). It should call functions and procedures located within the Algorithms package. It is called by RESOLVE_CONFLICTS.
- RESOLVE_CONFLICTS. This procedure should determine the resolution of any conflicts between two or more aircraft packages and between aircraft packages and surface-to-air missiles. It is called by RUN_SUPPORT_MISSIONS, RUN_STRIKE_MISSIONS, and RUN_AREA_MISSIONS. It in turn, calls the appropriate procedures and functions in the Algorithms package needed to resolve the type of conflict encountered.
- CONTINUE_MISSION. This procedure should continue an area mission once all the strike missions are finished and if it is the right period for the area mission to accomplish its mission. It is called by CHECK_AREA_MISSIONS.
- RUN_AREA_MISSIONS. This procedure should put all the area missions in their appropriate hexes. It should be processed before the strike missions are run so that the strike missions have the possibility of air conflicts. It is called by PERFORM_MISSIONS. It calls the Algorithms.DETERMINE_DIRECTION procedure to determine the movement of the aircraft package. It also calls AirHex.UPDATE_LOCATION to actually reflect the new location of the aircraft package. It also calls DETERMINE_TARGETS or RESOLVE_CONFLICTS depending on the aircraft package's mission.

- CHECK_AREA_MISSIONS. This procedure should detemine whether an area mission should be continued. It is called by PERFORM_MISSIONS and should, in turn, call the CONTINUE_MISSION and AircraftPackage.BREAKUP_ACPKG procedures.
- RUN_STRIKE_MISSIONS. This procedure should perform the strike missions. It is called by PERFORM_MISSIONS. It calls the Algorithms.DETERMINE_DIRECTION procedure to determine the movement of the aircraft package. It also calls AirHex.UPDATE_LOCATION to actually reflect the new location of the aircraft package and RESOLVE_CONFLICTS to resolve any conflicts it encounters. In addition, it calls the ACCOMPLISH_STRIKE_MISSION, JETTISON_WEAPONS, and AircraftPackage.BREAKUP_ACPKG procedures.
- RUN_SUPPORT_MISSIONS. This procedure should perform the support missions. It is called by PERFORM_MISSIONS. It calls the Algorithms.DETERMINE_DIRECTION procedure to determine the movement of the aircraft package. It also calls AirHex.UPDATE_LOCATION to actually reflect the new location of the aircraft package and RESOLVE_CONFLICTS to resolve any conflicts it encounters. In addition, it calls the ACCOMPLISH_SUPPORT_MISSION and AircraftPackage.BREAKUP_ACPKG procedures.

- AirHex. The package body does not contain any additional procedures or functions.
- Aircraft.
  - GET_PCL. This procedure is called by GET_AC and reads in from disk the preferred conventional load information for each type of aircraft.
  - GET_PBL. This procedure is called by GET_AC and reads in from disk the preferred biological/chemical load information for each type of aircraft.
  - GET_PNL. This procedure is called by GET_AC and reads in from disk the preferred nuclear load information for each type of aircraft.
- AircraftPackage.
  - GET_TARGETS. This procedure reads in the target number from the disk file.

- GET_ACPKG_AC. This procedure reads in the types and quantities of aircraft needed for the aircraft package, as well as their missions.
- WRITE_TARGETS. This procedure writes to disk the aircraft package's target number.
- WRITE_ACPKG_AC. This procedure writes out to disk the types and quantities of aircraft needed for the aircraft package, as well as their missions.

- AirHex. The package body does not contain any additional procedures or functions.
- Algorithms.
  - MOVE_DOWN_RIGHT. This procedure determines the new x and y air hex coordinates for an aircraft package moving down and right. It is called by GO_SE.
  - MOVE_LEFT_EVEN. This procedure determines the new x and y air hex coordinates for an aircraft package moving left when the starting x value is even. It is called by GO_SW and GO_NW.
  - MOVE_LEFT_ODD. This procedure determines the new x and y air hex coordinates for an aircraft package moving left when the starting x value is odd. It is called by GO_SW and GO_NW.
  - MOVE_DOWN_LEFT_EVEN. This procedure determines the new x and y air hex coordinates for an aircraft package moving down and left when the starting x value is even. It is called by GO_SW.
  - MOVE_DOWN_LEFT_ODD. This procedure determines the new x and y air hex coordinates for an aircraft package moving down and left when the starting x value is odd. It is called by GO_SW.
  - MOVE_RIGHT_EVEN. This procedure determines the new x and y air hex coordinates for an aircraft package moving right when the starting x value is even. It is called by GO_NE, GO_SE, and GO_NW.

- MOVE_RIGHT_ODD. This procedure determines the new x and y air hex coordinates for an aircraft package moving right when the starting x value is odd. It is called by GO_NE, GO_SE, and GO_NW.

- MOVE_UP_RIGHT_EVEN. This procedure determines the new x and y air hex coordinates for an aircraft package moving up and right when the starting x value is even. It is called by GO_NE and GO_NW.

- MOVE_UP_RIGHT_ODD. This procedure determines the new x and y air hex coordinates for an aircraft package moving up and right when the starting x value is odd. It is called by GO_NE and GO_NW.

- MOVE_UP_LEFT. This procedure determines the new x and y air hex coordinates for an aircraft package moving up and to the left. It is called by GO_NW.

- GO_SOUTH. This procedure is called by DETERMINE_DIRECTION. It is used if the destination hex is south of the current hex. It determines the hex number to which the aircraft package should move. It does not call any other procedure.

- GO_NORTH. This procedure is called by DETERMINE_DIRECTION. It is used if the destination hex is north of the current hex. It determines the hex number to which the aircraft package should move. It does not call any other procedure.

- GO_NE. This procedure is called by DETERMINE_DIRECTION. It is used if the destination hex is north-east of the current hex. Based on the current hex number and the destination hex number it, in turn, calls one of the following procedures: MOVE_RIGHT_EVEN, MOVE_RIGHT_ODD, MOVE_UP_RIGHT_EVEN, or MOVE_UP_RIGHT_ODD.

- GO_SE. This procedure is called by DETERMINE_DIRECTION. It is used if the destination hex is south-east of the current hex. Based on the current hex number and the destination hex number it, in turn, calls one of the following procedures: MOVE_RIGHT_EVEN, MOVE_RIGHT_ODD, or MOVE_DOWN_RIGHT.

- GO_SW. This procedure is called by DETERMINE_DIRECTION. It is used if the destination hex is south-west of the current hex. Based on the current hex number and the destination hex number it, in turn, calls one of the following procedures: MOVE_LEFT_EVEN, MOVE_LEFT_ODD, MOVE_DOWN_LEFT_EVEN, or MOVE_DOWN_LEFT_ODD.
- GO_NW. This procedure is called by DETERMINE_DIRECTION. It is used if the destination hex is north-west of the current hex. Based on the current hex number and the destination hex number it, in turn, calls one of the following procedures: MOVE_UP_LEFT, MOVE_RIGHT_EVEN, MOVE_RIGHT_ODD, MOVE_UP_RIGHT_EVEN, MOVE_UP_RIGHT_ODD, MOVE_LEFT_EVEN, or MOVE_LEFT_ODD.
- CALC_RADAR_QUALITY. This function is called by CALC_LOCAL_DETECTION and CALC_NO_MISS_FIRED. It is not one of the algorithms defined by Mann but was discussed in Chapter 4. It determines the radar quality.

- ArmySim. All of the following procedures were written by Ness and described in his thesis.
  - DEPOT_LOG. This procedure is called by LOG_SPT.
  - OVERCOME_OBSTACLE. This procedure is called by LOG_SPT and MOVEMENT.
  - DETERMINE_ROUTE. This procedure is called by BORDER_TRANSITION.
  - SELECT_ROUTE. This procedure is called by BORDER_TRANSITION.
  - CALC_DIRECTION. This procedure is nested within SELECT_ROUTE.
  - UPDATE_UNIT_LOCATION. This procedure is called by BORDER_TRANSITION and WITHDRAW_UNIT.
  - BORDER_TRANSITION. This procedure is called by MANEUVER.
  - MANEUVER. This procedure is called by MOVEMENT.
  - ARMY_INTEL. This procedure is called by INTELLIGENCE.

- WR_DESTROYED. This procedure is nested within WRITE_DATA.
- WR_GRID. This procedure is nested within WRITE_DATA.
- WR_UNITS. This procedure is nested within WRITE_DATA.
- WR_INTEL. This procedure is nested within WRITE_DATA.
- ASSESS_FP. This procedure is called by SET_UP.
- SET_ATK. This procedure is called by SET_UP.
- DESTROY. This procedure is called by ATTRITION.
- WITHDRAW_UNIT. This procedure is called by ATTRITION.

- Bases.
  - GET_RUNWAYS. This procedure is called by GET_BASES. It should read in from disk the information on the runways for a specific airbase.
  - GET_ALTERNATE_BASES. This procedure is called by GET_BASES. It should read in from disk the information on alternate bases. This information is used if an aircraft returns from a mission and there is not enough runway left at the plane's home location for it to land.
  - WRITE_RUNWAYS. This procedure is called by WRITE_BASES. It should write to disk the information on the runways for a specific airbase.
- Clock. The package body does not contain any additional procedures or functions.
- Forces. The package body does not contain any additional procedures or functions.
- GroundHex.
  - GET_OBSTACLES. This procedure should read in from disk the information on obstacles. It should be called by GET_HEX_SIDE.
  - GET_PIPELINES. This procedure should read in from disk the information on pipelines. It should be called by GET_HEX_SIDE.
  - GET_ROADS. This procedure should read in from disk the information on roads. It should be called by GET_HEX_SIDE.

- GET_RAILROADS. This procedure should read in from disk the information on railroads. It should be called by GET_HEX_SIDE.
- GET_RIVERS. This procedure should read in from disk the information on rivers. It should be called by GET_HEX_SIDE.
- GET_FEBA. This procedure should read in from disk the location of the FEBA. It should be called by GET_HEX_SIDE.
- GET_HEX_SIDE. This procedure should read in from disk the information on trafficability for the hex sides. It should be called by GET_GRID.
- WRITE_OBSTACLES. This procedure should write to disk the information on obstacles. It should be called by WRITE_HEX_SIDE.
- WRITE_PIPELINES. This procedure should write to disk the information on pipelines. It should be called by WRITE_HEX_SIDE.
- WRITE_ROADS. This procedure should write to disk the information on roads. It should be called by WRITE_HEX_SIDE.
- WRITE_RAILROADS. This procedure should write to disk the information on railroads. It should be called by WRITE_HEX_SIDE.
- WRITE_RIVERS. This procedure should write to disk the information on rivers. It should be called by WRITE_HEX_SIDE.
- WRITE_FEBA. This procedure should write to disk the location of the FEBA. It should be called by WRITE_HEX_SIDE.
- WRITE_HEX_SIDE. This procedure should write to disk the information on trafficability for the hex sides. It should be called by WRITE_GRID.

• GroundUnits.

- GET_SUPPORT_UNITS. This procedure should read in from disk the information on specific support units.
- GET_SUPPLY_MISSIONS. This procedure should read in from disk the data on the supply missions.

- GET_MISSIONS. This procedure should read in from disk the missions, or orders, the Army units are to perform.
- GET_OVERRIDE_MISSIONS. This procedure should read in from disk the override missions, or orders, which apply to specific Army units.
- WRITE_SUPPORT_UNITS. This procedure should write to disk the information on specific support units.
- WRITE_SUPPLY_MISSIONS. This procedure should write to disk the data on the supply missions.
- WRITE_MISSIONS. This procedure should write to disk the missions, or orders, the Army units are to perform.
- WRITE_OVERRIDE_MISSIONS. This procedure should write to disk the override missions, or orders, which apply to specific Army units.

- Hex.
  - *DETERMINE_WEATHER. This procedure determines the actual weather.* It is called by GET_WEATHER.
- Radars. The package body does not contain any additional procedures or functions.
- Satellites. The package body does not contain any additional procedures or functions.
- Targets. The package body does not contain any additional procedures or functions.
- Weapons. The package body does not contain any additional procedures or functions.

Bibliography

1. Booch, Grady. "Object-Oriented Development," *IEEE Transactions on Software Engineering*, *SE-12*:211–221 (February 1986).
2. Booch, Grady. *Software Components with Ada.* Menlo Park CA: The Benjamin/Cummings Publishing Company, Inc, 1987.
3. Booch, Grady. *Software Engineering with Ada* (Second Edition). Menlo Park CA: The Benjamin/Cummings Publishing Company, Inc, 1987.
4. Booch, Grady. *Object Oriented Design.* Redwood City CA: The Benjamin/Cummings Publishing Company, Inc, 1991.
5. Borrego, Jesus and others. "A space logistics simulation development in ADA." *Proceedings of the 1988 Winter Simulation Conference*, edited by Michael A. Abrams and others. 763–764. New York: IEEE Press, 1988.
6. Corporation, The MITRE, "A Preliminary Evaluation of Object-Oriented Programming for Ground Combat Modeling." Working Paper 83W00407, 1983.
7. Eldredge, David L. and others. "Applying the object-oriented paradigm to discrete event simulations using the C++ language," *Simulation*, *54*:83–91 (February 1990).
8. EVB Software Engineering, Inc. *An Object Oriented Design Handbook for Ada Software.* EVB Software Engineering, Inc, 1985.
9. Feldman, Michael B. *Data Structures with Ada.* Reston, Virginia: Reston Publishing Company, 1985.
10. Henderson-Sellers, Brian and Julian M. Edwards. "The Object-Oriented Systems Life Cycle," *Communications of the ACM*, *33*:142–159 (September 1990).
11. Horton, Andrew M. *Design and Implementation of a Graphical User Interface and a Database Management System for the Saber Theater-Level Wargame.* MS thesis, AFIT/GCS/ENG/91D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991 (DTIC # unknown).
12. Jean, Catherine and Alfred Strohmeier. "An experience in teaching OOD for ADA software," *Software Engineering Notes*, *15*:44–49 (October 1990).
13. Klabunde, Gary W. *An Animated Graphical Postprocessor for the Saber Wargame.* MS thesis, AFIT/GCS/ENG/91D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991 (DTIC # unknown).
14. Klabunde, Gary W. "History File." Report to Wargaming Research Group. AFIT, Wright-Patterson AFB OH, August 1991.
15. Korson, Tim and John D. McGregor. "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, *33*:40–60 (September 1990).
16. Krecker, Donald K. and Peter J. Lattimore, "An Integrated Coordinate System for Combat Modeling." Contract W-78-297-TR with BDM Corporation, 19 May 1978.

17. Mann, CPT William III. *Saber: A Theater Wargame.* MS thesis, AFIT/GOR/ENS/91M-09, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1990 (AD-A238825).

18. Melde, John E. and Philip G. Gage. "Ada simulation technology — methods and metrics," *Simulation*, *51*:57–69 (August 1988).

19. Ness, CPT Marlin A. "Maintenance and User's Manual for the Land Battle Program." Department of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1990.

20. Ness, CPT Marlin A. *A New Land Battle for the Theater War Exercise.* MS thesis, AFIT/GE/ENG/90J-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1990 (AD-A223087).

21. Powers, Wendy S. and Tom Nute. "Implementing a simulator as a set of Ada tasks." *Simulation in Ada*, edited by Brian Unger and others. 7–12. 1985.

22. Pritsker, A. Alan B. *Introduction to Simulation and SLAM II* (Third Edition). West Lafayette IN: Systems Publishing Corporation, 1986.

23. Roberts, Stephen D. and Joe Heim. "A perspective on object-oriented simulation." *Proceedings of the 1988 Winter Simulation Conference*, edited by Michael A. Abrams and others. 277–281. New York: IEEE Press, 1988.

24. Roth, Major Mark A., (Thesis advisor). Personal conversations. Electrical and Computer Engineering Department, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1 April through 20 September 1991.

25. Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," *Ada Letters*, *VII*:54–67 (July, August 1987).

26. Shore, Dr. R. W. "Discrete-Event Simulation in Ada: Concepts," *Ada Letters*, *VII*:105–112 (September, October 1987).

27. Shtern, Dr. Victor. "Testing of software for discrete simulation models in Ada," *Simulation in Ada*, 13–18 (1985).

28. Sommerville, Ian. *Software Engineering* (Third Edition). Wokingham England: Addison-Wesley Publishing Company, 1989.

29. Unger, Brian W. "Object oriented simulation — Ada, C++, Simula." *Proceedings of the 1986 Winter Simulation Conference*, edited by J. Wilson and others. 123–124. New York: IEEE Press, 1986.

30. Unger, Brian W. et al. *Simulation Software and Ada.* LaJolla CA: Simulation Councils, Inc., 1984.

31. Zeitak, G. and J. Arlan, editors. *Missiles & Missile Systems: An International Directory.* Rehovot, Israel. A to Z Independent Information Indexing, 1987.